Sixth Edition

John L. Hennessy | David A. Patterson

COMPUTER ARCHITECTURE

A Quantitative Approach

MK
MORGAN KAUFMANN

VISIT…

LANZAROTE
Caliente.com

Computer Architecture Formulas

1. *CPU time* = Instruction count × Clock cycles per instruction × Clock cycle time
2. X is n times faster than Y: $n = \text{Execution time}_Y / \text{Execution time}_X = \text{Performance}_X / \text{Performance}_Y$
3. *Amdahl's Law:* $\text{Speedup}_{\text{overall}} = \dfrac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \dfrac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$
4. $\mathit{Energy}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$
5. $\mathit{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$
6. $\mathit{Power}_{\text{static}} \propto \text{Current}_{\text{static}} \times \text{Voltage}$
7. *Availability* = Mean time to fail / (Mean time to fail + Mean time to repair)
8. $\mathit{Die\ yield} = \text{Wafer yield} \times 1 / (1 + \text{Defects per unit area} \times \text{Die area})^N$

   where Wafer yield accounts for wafers that are so bad they need not be tested and N is a parameter called the process-complexity factor, a measure of manufacturing difficulty. N ranges from 11.5 to 15.5 in 2011.
9. *Means—arithmetic (AM), weighted arithmetic (WAM), and geometric (GM):*

$$\text{AM} = \frac{1}{n}\sum_{i=1}^{n} \text{Time}_i \quad \text{WAM} = \sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i \quad \text{GM} = \sqrt[n]{\prod_{i=1}^{n} \text{Time}_i}$$

   where Time_i is the execution time for the ith program of a total of n in the workload, Weight_i is the weighting of the ith program in the workload.
10. *Average memory-access time* = Hit time + Miss rate × Miss penalty
11. *Misses per instruction* = Miss rate × Memory access per instruction
12. *Cache index size:* $2^{\text{index}} = \text{Cache size} / (\text{Block size} \times \text{Set associativity})$
13. *Power Utilization Effectiveness* (*PUE*) of a Warehouse Scale Computer = $\dfrac{\text{Total Facility Power}}{\text{IT Equipment Power}}$

Rules of Thumb

1. *Amdahl/Case Rule:* A balanced computer system needs about 1 MB of main memory capacity and 1 megabit per second of I/O bandwidth per MIPS of CPU performance.
2. *90/10 Locality Rule:* A program executes about 90% of its instructions in 10% of its code.
3. *Bandwidth Rule:* Bandwidth grows by at least the square of the improvement in latency.
4. *2:1 Cache Rule:* The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size $N/2$.
5. *Dependability Rule:* Design with no single point of failure.
6. *Watt-Year Rule*: The fully burdened cost of a Watt per year in a Warehouse Scale Computer in North America in 2011, including the cost of amortizing the power and cooling infrastructure, is about $2.

In Praise of *Computer Architecture: A Quantitative Approach* Sixth Edition

"Although important concepts of architecture are timeless, this edition has been thoroughly updated with the latest technology developments, costs, examples, and references. Keeping pace with recent developments in open-sourced architecture, the instruction set architecture used in the book has been updated to use the RISC-V ISA."

—from the foreword by Norman P. Jouppi, Google

"*Computer Architecture: A Quantitative Approach* is a classic that, like fine wine, just keeps getting better. I bought my first copy as I finished up my undergraduate degree and it remains one of my most frequently referenced texts today."

—James Hamilton, Amazon Web Service

"Hennessy and Patterson wrote the first edition of this book when graduate students built computers with 50,000 transistors. Today, warehouse-size computers contain that many servers, each consisting of dozens of independent processors and billions of transistors. The evolution of computer architecture has been rapid and relentless, but *Computer Architecture: A Quantitative Approach* has kept pace, with each edition accurately explaining and analyzing the important emerging ideas that make this field so exciting."

—James Larus, Microsoft Research

"Another timely and relevant update to a classic, once again also serving as a window into the relentless and exciting evolution of computer architecture! The new discussions in this edition on the slowing of Moore's law and implications for future systems are must-reads for both computer architects and practitioners working on broader systems."

—Parthasarathy (Partha) Ranganathan, Google

"I love the 'Quantitative Approach' books because they are written by engineers, for engineers. John Hennessy and Dave Patterson show the limits imposed by mathematics and the possibilities enabled by materials science. Then they teach through real-world examples how architects analyze, measure, and compromise to build working systems. This sixth edition comes at a critical time: Moore's Law is fading just as deep learning demands unprecedented compute cycles. The new chapter on domain-specific architectures documents a number of promising approaches and prophesies a rebirth in computer architecture. Like the scholars of the European Renaissance, computer architects must understand our own history, and then combine the lessons of that history with new techniques to remake the world."

—Cliff Young, Google

This page intentionally left blank

Computer Architecture

A Quantitative Approach

Sixth Edition

John L. Hennessy is a Professor of Electrical Engineering and Computer Science at Stanford University, where he has been a member of the faculty since 1977 and was, from 2000 to 2016, its 10th President. He currently serves as the Director of the Knight-Hennessy Fellowship, which provides graduate fellowships to potential future leaders. Hennessy is a Fellow of the IEEE and ACM, a member of the National Academy of Engineering, the National Academy of Science, and the American Philosophical Society, and a Fellow of the American Academy of Arts and Sciences. Among his many awards are the 2001 Eckert-Mauchly Award for his contributions to RISC technology, the 2001 Seymour Cray Computer Engineering Award, and the 2000 John von Neumann Award, which he shared with David Patterson. He has also received 10 honorary doctorates.

In 1981, he started the MIPS project at Stanford with a handful of graduate students. After completing the project in 1984, he took a leave from the university to cofound MIPS Computer Systems, which developed one of the first commercial RISC microprocessors. As of 2017, over 5 billion MIPS microprocessors have been shipped in devices ranging from video games and palmtop computers to laser printers and network switches. Hennessy subsequently led the DASH (Director Architecture for Shared Memory) project, which prototyped the first scalable cache coherent multiprocessor; many of the key ideas have been adopted in modern multiprocessors. In addition to his technical activities and university responsibilities, he has continued to work with numerous start-ups, both as an early-stage advisor and an investor.

David A. Patterson became a Distinguished Engineer at Google in 2016 after 40 years as a UC Berkeley professor. He joined UC Berkeley immediately after graduating from UCLA. He still spends a day a week in Berkeley as an Emeritus Professor of Computer Science. His teaching has been honored by the Distinguished Teaching Award from the University of California, the Karlstrom Award from ACM, and the Mulligan Education Medal and Undergraduate Teaching Award from IEEE. Patterson received the IEEE Technical Achievement Award and the ACM Eckert-Mauchly Award for contributions to RISC, and he shared the IEEE Johnson Information Storage Award for contributions to RAID. He also shared the IEEE John von Neumann Medal and the C & C Prize with John Hennessy. Like his co-author, Patterson is a Fellow of the American Academy of Arts and Sciences, the Computer History Museum, ACM, and IEEE, and he was elected to the National Academy of Engineering, the National Academy of Sciences, and the Silicon Valley Engineering Hall of Fame. He served on the Information Technology Advisory Committee to the President of the United States, as chair of the CS division in the Berkeley EECS department, as chair of the Computing Research Association, and as President of ACM. This record led to Distinguished Service Awards from ACM, CRA, and SIGARCH. He is currently Vice-Chair of the Board of Directors of the RISC-V Foundation.

At Berkeley, Patterson led the design and implementation of RISC I, likely the first VLSI reduced instruction set computer, and the foundation of the commercial SPARC architecture. He was a leader of the Redundant Arrays of Inexpensive Disks (RAID) project, which led to dependable storage systems from many companies. He was also involved in the Network of Workstations (NOW) project, which led to cluster technology used by Internet companies and later to cloud computing. His current interests are in designing domain-specific architectures for machine learning, spreading the word on the open RISC-V instruction set architecture, and in helping the UC Berkeley RISELab (Real-time Intelligent Secure Execution).

Computer Architecture
A Quantitative Approach

Sixth Edition

John L. Hennessy
Stanford University

David A. Patterson
University of California, Berkeley

With Contributions by

Krste Asanović
University of California, Berkeley
Jason D. Bakos
University of South Carolina
Robert P. Colwell
R&E Colwell & Assoc. Inc.
Abhishek Bhattacharjee
Rutgers University
Thomas M. Conte
Georgia Tech
José Duato
Proemisa
Diana Franklin
University of Chicago
David Goldberg
eBay
Norman P. Jouppi
Google
Sheng Li
Intel Labs
Naveen Muralimanohar
HP Labs
Gregory D. Peterson
University of Tennessee
Timothy M. Pinkston
University of Southern California
Parthasarathy Ranganathan
Google
David A. Wood
University of Wisconsin–Madison
Cliff Young
Google
Amr Zaky
University of Santa Clara

ELSEVIER

MK
MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

© 2019 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-811905-1

For information on all Morgan Kaufmann publications
visit our website at https://www.elsevier.com/books-and-journals

ELSEVIER
Book Aid International
Working together
to grow libraries in
developing countries
www.elsevier.com • www.bookaid.org

Publisher: Katey Birtcher
Acquisition Editor: Stephen Merken
Developmental Editor: Nate McFadden
Production Project Manager: Stalin Viswanathan
Cover Designer: Christian J. Bilbow

Typeset by SPi Global, India

To Andrea, Linda, and our four sons

Foreword

by Norman P. Jouppi, Google

Much of the improvement in computer performance over the last 40 years has been provided by computer architecture advancements that have leveraged Moore's Law and Dennard scaling to build larger and more parallel systems. Moore's Law is the observation that the maximum number of transistors in an integrated circuit doubles approximately every two years. Dennard scaling refers to the reduction of MOS supply voltage in concert with the scaling of feature sizes, so that as transistors get smaller, their power density stays roughly constant. With the end of Dennard scaling a decade ago, and the recent slowdown of Moore's Law due to a combination of physical limitations and economic factors, the sixth edition of the preeminent textbook for our field couldn't be more timely. Here are some reasons.

First, because domain-specific architectures can provide equivalent performance and power benefits of three or more historical generations of Moore's Law and Dennard scaling, they now can provide better implementations than may ever be possible with future scaling of general-purpose architectures. And with the diverse application space of computers today, there are many potential areas for architectural innovation with domain-specific architectures. Second, high-quality implementations of open-source architectures now have a much longer lifetime due to the slowdown in Moore's Law. This gives them more opportunities for continued optimization and refinement, and hence makes them more attractive. Third, with the slowing of Moore's Law, different technology components have been scaling heterogeneously. Furthermore, new technologies such as 2.5D stacking, new nonvolatile memories, and optical interconnects have been developed to provide more than Moore's Law can supply alone. To use these new technologies and nonhomogeneous scaling effectively, fundamental design decisions need to be reexamined from first principles. Hence it is important for students, professors, and practitioners in the industry to be skilled in a wide range of both old and new architectural techniques. All told, I believe this is the most exciting time in computer architecture since the industrial exploitation of instruction-level parallelism in microprocessors 25 years ago.

The largest change in this edition is the addition of a new chapter on domain-specific architectures. It's long been known that customized domain-specific architectures can have higher performance, lower power, and require less silicon area than general-purpose processor implementations. However when general-purpose processors were increasing in single-threaded performance by 40% per year (see Fig. 1.11), the extra time to market required to develop a custom architecture vs. using a leading-edge standard microprocessor could cause the custom architecture to lose much of its advantage. In contrast, today single-core performance is improving very slowly, meaning that the benefits of custom architectures will not be made obsolete by general-purpose processors for a very long time, if ever. Chapter 7 covers several domain-specific architectures. Deep neural networks have very high computation requirements but lower data precision requirements – this combination can benefit significantly from custom architectures. Two example architectures and implementations for deep neural networks are presented: one optimized for inference and a second optimized for training. Image processing is another example domain; it also has high computation demands and benefits from lower-precision data types. Furthermore, since it is often found in mobile devices, the power savings from custom architectures are also very valuable. Finally, by nature of their reprogrammability, FPGA-based accelerators can be used to implement a variety of different domain-specific architectures on a single device. They also can benefit more irregular applications that are frequently updated, like accelerating internet search.

Although important concepts of architecture are timeless, this edition has been thoroughly updated with the latest technology developments, costs, examples, and references. Keeping pace with recent developments in open-sourced architecture, the instruction set architecture used in the book has been updated to use the RISC-V ISA.

On a personal note, after enjoying the privilege of working with John as a graduate student, I am now enjoying the privilege of working with Dave at Google. What an amazing duo!

Contents

Foreword **ix**

Preface **xvii**

Acknowledgments **xxv**

Chapter 1 Fundamentals of Quantitative Design and Analysis

1.1 Introduction 2
1.2 Classes of Computers 6
1.3 Defining Computer Architecture 11
1.4 Trends in Technology 18
1.5 Trends in Power and Energy in Integrated Circuits 23
1.6 Trends in Cost 29
1.7 Dependability 36
1.8 Measuring, Reporting, and Summarizing Performance 39
1.9 Quantitative Principles of Computer Design 48
1.10 Putting It All Together: Performance, Price, and Power 55
1.11 Fallacies and Pitfalls 58
1.12 Concluding Remarks 64
1.13 Historical Perspectives and References 67
Case Studies and Exercises by Diana Franklin 67

Chapter 2 Memory Hierarchy Design

2.1 Introduction 78
2.2 Memory Technology and Optimizations 84
2.3 Ten Advanced Optimizations of Cache Performance 94
2.4 Virtual Memory and Virtual Machines 118
2.5 Cross-Cutting Issues: The Design of Memory Hierarchies 126
2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700 129
2.7 Fallacies and Pitfalls 142
2.8 Concluding Remarks: Looking Ahead 146
2.9 Historical Perspectives and References 148

Case Studies and Exercises by Norman P. Jouppi, Rajeev Balasubramonian, Naveen Muralimanohar, and Sheng Li 148

Chapter 3 Instruction-Level Parallelism and Its Exploitation

3.1 Instruction-Level Parallelism: Concepts and Challenges 168
3.2 Basic Compiler Techniques for Exposing ILP 176
3.3 Reducing Branch Costs With Advanced Branch Prediction 182
3.4 Overcoming Data Hazards With Dynamic Scheduling 191
3.5 Dynamic Scheduling: Examples and the Algorithm 201
3.6 Hardware-Based Speculation 208
3.7 Exploiting ILP Using Multiple Issue and Static Scheduling 218
3.8 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation 222
3.9 Advanced Techniques for Instruction Delivery and Speculation 228
3.10 Cross-Cutting Issues 240
3.11 Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput 242
3.12 Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53 247
3.13 Fallacies and Pitfalls 258
3.14 Concluding Remarks: What's Ahead? 264
3.15 Historical Perspective and References 266
Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell 266

Chapter 4 Data-Level Parallelism in Vector, SIMD, and GPU Architectures

4.1 Introduction 282
4.2 Vector Architecture 283
4.3 SIMD Instruction Set Extensions for Multimedia 304
4.4 Graphics Processing Units 310
4.5 Detecting and Enhancing Loop-Level Parallelism 336
4.6 Cross-Cutting Issues 345
4.7 Putting It All Together: Embedded Versus Server GPUs and Tesla Versus Core i7 346
4.8 Fallacies and Pitfalls 353
4.9 Concluding Remarks 357
4.10 Historical Perspective and References 357
Case Study and Exercises by Jason D. Bakos 357

Chapter 5 Thread-Level Parallelism

5.1 Introduction 368
5.2 Centralized Shared-Memory Architectures 377
5.3 Performance of Symmetric Shared-Memory Multiprocessors 393

5.4 Distributed Shared-Memory and Directory-Based Coherence 404
5.5 Synchronization: The Basics 412
5.6 Models of Memory Consistency: An Introduction 417
5.7 Cross-Cutting Issues 422
5.8 Putting It All Together: Multicore Processors and Their Performance 426
5.9 Fallacies and Pitfalls 438
5.10 The Future of Multicore Scaling 442
5.11 Concluding Remarks 444
5.12 Historical Perspectives and References 445
Case Studies and Exercises by Amr Zaky and David A. Wood 446

Chapter 6 **Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism**

6.1 Introduction 466
6.2 Programming Models and Workloads for Warehouse-Scale Computers 471
6.3 Computer Architecture of Warehouse-Scale Computers 477
6.4 The Efficiency and Cost of Warehouse-Scale Computers 482
6.5 Cloud Computing: The Return of Utility Computing 490
6.6 Cross-Cutting Issues 501
6.7 Putting It All Together: A Google Warehouse-Scale Computer 503
6.8 Fallacies and Pitfalls 514
6.9 Concluding Remarks 518
6.10 Historical Perspectives and References 519
Case Studies and Exercises by Parthasarathy Ranganathan 519

Chapter 7 **Domain-Specific Architectures**

7.1 Introduction 540
7.2 Guidelines for DSAs 543
7.3 Example Domain: Deep Neural Networks 544
7.4 Google's Tensor Processing Unit, an Inference Data Center Accelerator 557
7.5 Microsoft Catapult, a Flexible Data Center Accelerator 567
7.6 Intel Crest, a Data Center Accelerator for Training 579
7.7 Pixel Visual Core, a Personal Mobile Device Image Processing Unit 579
7.8 Cross-Cutting Issues 592
7.9 Putting It All Together: CPUs Versus GPUs Versus DNN Accelerators 595
7.10 Fallacies and Pitfalls 602
7.11 Concluding Remarks 604
7.12 Historical Perspectives and References 606
Case Studies and Exercises by Cliff Young 606

Appendix A Instruction Set Principles

A.1 Introduction A-2
A.2 Classifying Instruction Set Architectures A-3
A.3 Memory Addressing A-7
A.4 Type and Size of Operands A-13
A.5 Operations in the Instruction Set A-15
A.6 Instructions for Control Flow A-16
A.7 Encoding an Instruction Set A-21
A.8 Cross-Cutting Issues: The Role of Compilers A-24
A.9 Putting It All Together: The RISC-V Architecture A-33
A.10 Fallacies and Pitfalls A-42
A.11 Concluding Remarks A-46
A.12 Historical Perspective and References A-47
Exercises by Gregory D. Peterson A-47

Appendix B Review of Memory Hierarchy

B.1 Introduction B-2
B.2 Cache Performance B-15
B.3 Six Basic Cache Optimizations B-22
B.4 Virtual Memory B-40
B.5 Protection and Examples of Virtual Memory B-49
B.6 Fallacies and Pitfalls B-57
B.7 Concluding Remarks B-59
B.8 Historical Perspective and References B-59
Exercises by Amr Zaky B-60

Appendix C Pipelining: Basic and Intermediate Concepts

C.1 Introduction C-2
C.2 The Major Hurdle of Pipelining—Pipeline Hazards C-10
C.3 How Is Pipelining Implemented? C-26
C.4 What Makes Pipelining Hard to Implement? C-37
C.5 Extending the RISC V Integer Pipeline to Handle Multicycle Operations C-45
C.6 Putting It All Together: The MIPS R4000 Pipeline C-55
C.7 Cross-Cutting Issues C-65
C.8 Fallacies and Pitfalls C-70
C.9 Concluding Remarks C-71
C.10 Historical Perspective and References C-71
Updated Exercises by Diana Franklin C-71

Online Appendices

Appendix D **Storage Systems**

Appendix E **Embedded Systems**
by Thomas M. Conte

Appendix F **Interconnection Networks**
by Timothy M. Pinkston and José Duato

Appendix G **Vector Processors in More Depth**
by Krste Asanovic

Appendix H **Hardware and Software for VLIW and EPIC**

Appendix I **Large-Scale Multiprocessors and Scientific Applications**

Appendix J **Computer Arithmetic**
by David Goldberg

Appendix K **Survey of Instruction Set Architectures**

Appendix L **Advanced Concepts on Address Translation**
by Abhishek Bhattacharjee

Appendix M **Historical Perspectives and References**

References R-1

Index I-1

This page intentionally left blank

Preface

Why We Wrote This Book

Through six editions of this book, our goal has been to describe the basic principles underlying what will be tomorrow's technological developments. Our excitement about the opportunities in computer architecture has not abated, and we echo what we said about the field in the first edition: "It is not a dreary science of paper machines that will never work. No! It's a discipline of keen intellectual interest, requiring the balance of marketplace forces to cost-performance-power, leading to glorious failures and some notable successes."

Our primary objective in writing our first book was to change the way people learn and think about computer architecture. We feel this goal is still valid and important. The field is changing daily and must be studied with real examples and measurements on real computers, rather than simply as a collection of definitions and designs that will never need to be realized. We offer an enthusiastic welcome to anyone who came along with us in the past, as well as to those who are joining us now. Either way, we can promise the same quantitative approach to, and analysis of, real systems.

As with earlier versions, we have strived to produce a new edition that will continue to be as relevant for professional engineers and architects as it is for those involved in advanced computer architecture and design courses. Like the first edition, this edition has a sharp focus on new platforms—personal mobile devices and warehouse-scale computers—and new architectures—specifically, domain-specific architectures. As much as its predecessors, this edition aims to demystify computer architecture through an emphasis on cost-performance-energy trade-offs and good engineering design. We believe that the field has continued to mature and move toward the rigorous quantitative foundation of long-established scientific and engineering disciplines.

This Edition

The ending of Moore's Law and Dennard scaling is having as profound effect on computer architecture as did the switch to multicore. We retain the focus on the extremes in size of computing, with personal mobile devices (PMDs) such as cell phones and tablets as the clients and warehouse-scale computers offering cloud computing as the server. We also maintain the other theme of parallelism in all its forms: *data-level parallelism (DLP)* in Chapters 1 and 4, *instruction-level parallelism (ILP)* in Chapter 3, *thread-level parallelism* in Chapter 5, and *request-level parallelism* (RLP) in Chapter 6.

The most pervasive change in this edition is switching from MIPS to the RISC-V instruction set. We suspect this modern, modular, open instruction set may become a significant force in the information technology industry. It may become as important in computer architecture as Linux is for operating systems.

The newcomer in this edition is Chapter 7, which introduces domain-specific architectures with several concrete examples from industry.

As before, the first three appendices in the book give basics on the RISC-V instruction set, memory hierarchy, and pipelining for readers who have not read a book like *Computer Organization and Design*. To keep costs down but still supply supplemental material that is of interest to some readers, available online at https://www.elsevier.com/books-and-journals/book-companion/9780128119051 are nine more appendices. There are more pages in these appendices than there are in this book!

This edition continues the tradition of using real-world examples to demonstrate the ideas, and the "Putting It All Together" sections are brand new. The "Putting It All Together" sections of this edition include the pipeline organizations and memory hierarchies of the ARM Cortex A8 processor, the Intel core i7 processor, the NVIDIA GTX-280 and GTX-480 GPUs, and one of the Google warehouse-scale computers.

Topic Selection and Organization

As before, we have taken a conservative approach to topic selection, for there are many more interesting ideas in the field than can reasonably be covered in a treatment of basic principles. We have steered away from a comprehensive survey of every architecture a reader might encounter. Instead, our presentation focuses on core concepts likely to be found in any new machine. The key criterion remains that of selecting ideas that have been examined and utilized successfully enough to permit their discussion in quantitative terms.

Our intent has always been to focus on material that is not available in equivalent form from other sources, so we continue to emphasize advanced content wherever possible. Indeed, there are several systems here whose descriptions cannot be found in the literature. (Readers interested strictly in a more basic introduction to computer architecture should read *Computer Organization and Design: The Hardware/Software Interface*.)

An Overview of the Content

Chapter 1 includes formulas for energy, static power, dynamic power, integrated circuit costs, reliability, and availability. (These formulas are also found on the front inside cover.) Our hope is that these topics can be used through the rest of the book. In addition to the classic quantitative principles of computer design and performance measurement, it shows the slowing of performance improvement of general-purpose microprocessors, which is one inspiration for domain-specific architectures.

Our view is that the instruction set architecture is playing less of a role today than in 1990, so we moved this material to Appendix A. It now uses the RISC-V architecture. (For quick review, a summary of the RISC-V ISA can be found on the back inside cover.) For fans of ISAs, Appendix K was revised for this edition and covers 8 RISC architectures (5 for desktop and server use and 3 for embedded use), the 80×86, the DEC VAX, and the IBM 360/370.

We then move onto memory hierarchy in Chapter 2, since it is easy to apply the cost-performance-energy principles to this material, and memory is a critical resource for the rest of the chapters. As in the past edition, Appendix B contains an introductory review of cache principles, which is available in case you need it. Chapter 2 discusses 10 advanced optimizations of caches. The chapter includes virtual machines, which offer advantages in protection, software management, and hardware management, and play an important role in cloud computing. In addition to covering SRAM and DRAM technologies, the chapter includes new material both on Flash memory and on the use of stacked die packaging for extending the memory hierarchy. The PIAT examples are the ARM Cortex A8, which is used in PMDs, and the Intel Core i7, which is used in servers.

Chapter 3 covers the exploitation of instruction-level parallelism in high-performance processors, including superscalar execution, branch prediction (including the new tagged hybrid predictors), speculation, dynamic scheduling, and simultaneous multithreading. As mentioned earlier, Appendix C is a review of pipelining in case you need it. Chapter 3 also surveys the limits of ILP. Like Chapter 2, the PIAT examples are again the ARM Cortex A8 and the Intel Core i7. While the third edition contained a great deal on Itanium and VLIW, this material is now in Appendix H, indicating our view that this architecture did not live up to the earlier claims.

The increasing importance of multimedia applications such as games and video processing has also increased the importance of architectures that can exploit data level parallelism. In particular, there is a rising interest in computing using graphical processing units (GPUs), yet few architects understand how GPUs really work. We decided to write a new chapter in large part to unveil this new style of computer architecture. Chapter 4 starts with an introduction to vector architectures, which acts as a foundation on which to build explanations of multimedia SIMD instruction set extensions and GPUs. (Appendix G goes into even more depth on vector architectures.) This chapter introduces the Roofline performance model and then uses it to compare the Intel Core i7 and the NVIDIA GTX 280 and GTX 480 GPUs. The chapter also describes the Tegra 2 GPU for PMDs.

Chapter 5 describes multicore processors. It explores symmetric and distributed-memory architectures, examining both organizational principles and performance. The primary additions to this chapter include more comparison of multicore organizations, including the organization of multicore-multilevel caches, multicore coherence schemes, and on-chip multicore interconnect. Topics in synchronization and memory consistency models are next. The example is the Intel Core i7. Readers interested in more depth on interconnection networks should read Appendix F, and those interested in larger scale multiprocessors and scientific applications should read Appendix I.

Chapter 6 describes warehouse-scale computers (WSCs). It was extensively revised based on help from engineers at Google and Amazon Web Services. This chapter integrates details on design, cost, and performance of WSCs that few architects are aware of. It starts with the popular MapReduce programming model before describing the architecture and physical implementation of WSCs, including cost. The costs allow us to explain the emergence of cloud computing, whereby it can be cheaper to compute using WSCs in the cloud than in your local datacenter. The PIAT example is a description of a Google WSC that includes information published for the first time in this book.

The new Chapter 7 motivates the need for Domain-Specific Architectures (DSAs). It draws guiding principles for DSAs based on the four examples of DSAs. Each DSA corresponds to chips that have been deployed in commercial settings. We also explain why we expect a renaissance in computer architecture via DSAs given that single-thread performance of general-purpose microprocessors has stalled.

This brings us to Appendices A through M. Appendix A covers principles of ISAs, including RISC-V, and Appendix K describes 64-bit versions of RISC V, ARM, MIPS, Power, and SPARC and their multimedia extensions. It also includes some classic architectures (80x86, VAX, and IBM 360/370) and popular embedded instruction sets (Thumb-2, microMIPS, and RISC V C). Appendix H is related, in that it covers architectures and compilers for VLIW ISAs.

As mentioned earlier, Appendix B and Appendix C are tutorials on basic caching and pipelining concepts. Readers relatively new to caching should read Appendix B before Chapter 2, and those new to pipelining should read Appendix C before Chapter 3.

Appendix D, "Storage Systems," has an expanded discussion of reliability and availability, a tutorial on RAID with a description of RAID 6 schemes, and rarely found failure statistics of real systems. It continues to provide an introduction to queuing theory and I/O performance benchmarks. We evaluate the cost, performance, and reliability of a real cluster: the Internet Archive. The "Putting It All Together" example is the NetApp FAS6000 filer.

Appendix E, by Thomas M. Conte, consolidates the embedded material in one place.

Appendix F, on interconnection networks, is revised by Timothy M. Pinkston and José Duato. Appendix G, written originally by Krste Asanović, includes a description of vector processors. We think these two appendices are some of the best material we know of on each topic.

Appendix H describes VLIW and EPIC, the architecture of Itanium.

Appendix I describes parallel processing applications and coherence protocols for larger-scale, shared-memory multiprocessing. Appendix J, by David Goldberg, describes computer arithmetic.

Appendix L, by Abhishek Bhattacharjee, is new and discusses advanced techniques for memory management, focusing on support for virtual machines and design of address translation for very large address spaces. With the growth in clouds processors, these architectural enhancements are becoming more important.

Appendix M collects the "Historical Perspective and References" from each chapter into a single appendix. It attempts to give proper credit for the ideas in each chapter and a sense of the history surrounding the inventions. We like to think of this as presenting the human drama of computer design. It also supplies references that the student of architecture may want to pursue. If you have time, we recommend reading some of the classic papers in the field that are mentioned in these sections. It is both enjoyable and educational to hear the ideas directly from the creators. "Historical Perspective" was one of the most popular sections of prior editions.

Navigating the Text

There is no single best order in which to approach these chapters and appendices, except that all readers should start with Chapter 1. If you don't want to read everything, here are some suggested sequences:

- *Memory Hierarchy:* Appendix B, Chapter 2, and Appendices D and M.
- *Instruction-Level Parallelism:* Appendix C, Chapter 3, and Appendix H
- *Data-Level Parallelism:* Chapters 4, 6, and 7, Appendix G
- *Thread-Level Parallelism:* Chapter 5, Appendices F and I
- *Request-Level Parallelism:* Chapter 6
- *ISA:* Appendices A and K

Appendix E can be read at any time, but it might work best if read after the ISA and cache sequences. Appendix J can be read whenever arithmetic moves you. You should read the corresponding portion of Appendix M after you complete each chapter.

Chapter Structure

The material we have selected has been stretched upon a consistent framework that is followed in each chapter. We start by explaining the ideas of a chapter. These ideas are followed by a "Crosscutting Issues" section, a feature that shows how the ideas covered in one chapter interact with those given in other chapters. This is followed by a "Putting It All Together" section that ties these ideas together by showing how they are used in a real machine.

Next in the sequence is "Fallacies and Pitfalls," which lets readers learn from the mistakes of others. We show examples of common misunderstandings and architectural traps that are difficult to avoid even when you know they are lying in wait for you. The "Fallacies and Pitfalls" sections is one of the most popular sections of the book. Each chapter ends with a "Concluding Remarks" section.

Case Studies With Exercises

Each chapter ends with case studies and accompanying exercises. Authored by experts in industry and academia, the case studies explore key chapter concepts and verify understanding through increasingly challenging exercises. Instructors should find the case studies sufficiently detailed and robust to allow them to create their own additional exercises.

Brackets for each exercise (<chapter.section>) indicate the text sections of primary relevance to completing the exercise. We hope this helps readers to avoid exercises for which they haven't read the corresponding section, in addition to providing the source for review. Exercises are rated, to give the reader a sense of the amount of time required to complete an exercise:

[10] Less than 5 min (to read and understand)
[15] 5–15 min for a full answer
[20] 15–20 min for a full answer
[25] 1 h for a full written answer
[30] Short programming project: less than 1 full day of programming
[40] Significant programming project: 2 weeks of elapsed time
[Discussion] Topic for discussion with others

Solutions to the case studies and exercises are available for instructors who register at *textbooks.elsevier.com*.

Supplemental Materials

A variety of resources are available online at https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1, including the following:

- Reference appendices, some guest authored by subject experts, covering a range of advanced topics
- Historical perspectives material that explores the development of the key ideas presented in each of the chapters in the text

- Instructor slides in PowerPoint
- Figures from the book in PDF, EPS, and PPT formats
- Links to related material on the Web
- List of errata

New materials and links to other resources available on the Web will be added on a regular basis.

Helping Improve This Book

Finally, it is possible to make money while reading this book. (Talk about cost performance!) If you read the Acknowledgments that follow, you will see that we went to great lengths to correct mistakes. Since a book goes through many printings, we have the opportunity to make even more corrections. If you uncover any remaining resilient bugs, please contact the publisher by electronic mail (ca6bugs@mkp.com).

We welcome general comments to the text and invite you to send them to a separate email address at ca6comments@mkp.com.

Concluding Remarks

Once again, this book is a true co-authorship, with each of us writing half the chapters and an equal share of the appendices. We can't imagine how long it would have taken without someone else doing half the work, offering inspiration when the task seemed hopeless, providing the key insight to explain a difficult concept, supplying over-the-weekend reviews of chapters, and commiserating when the weight of our other obligations made it hard to pick up the pen.

Thus, once again, we share equally the blame for what you are about to read.

John Hennessy ■ *David Patterson*

This page intentionally left blank

Acknowledgments

Although this is only the sixth edition of this book, we have actually created ten different versions of the text: three versions of the first edition (alpha, beta, and final) and two versions of the second, third, and fourth editions (beta and final). Along the way, we have received help from hundreds of reviewers and users. Each of these people has helped make this book better. Thus, we have chosen to list all of the people who have made contributions to some version of this book.

Contributors to the Sixth Edition

Like prior editions, this is a community effort that involves scores of volunteers. Without their help, this edition would not be nearly as polished.

Reviewers

Jason D. Bakos, University of South Carolina; Rajeev Balasubramonian, University of Utah; Jose Delgado-Frias, Washington State University; Diana Franklin, The University of Chicago; Norman P. Jouppi, Google; Hugh C. Lauer, Worcester Polytechnic Institute; Gregory Peterson, University of Tennessee; Bill Pierce, Hood College; Parthasarathy Ranganathan, Google; William H. Robinson, Vanderbilt University; Pat Stakem, Johns Hopkins University; Cliff Young, Google; Amr Zaky, University of Santa Clara; Gerald Zarnett, Ryerson University; Huiyang Zhou, North Carolina State University.

Members of the University of California-Berkeley Par Lab and RAD Lab who gave frequent reviews of Chapters 1, 4, and 6 and shaped the explanation of GPUs and WSCs: Krste Asanović, Michael Armbrust, Scott Beamer, Sarah Bird, Bryan Catanzaro, Jike Chong, Henry Cook, Derrick Coetzee, Randy Katz, Yunsup Lee, Leo Meyervich, Mark Murphy, Zhangxi Tan, Vasily Volkov, and Andrew Waterman.

Appendices

Krste Asanović, University of California, Berkeley (Appendix G); Abhishek Bhattacharjee, Rutgers University (Appendix L); Thomas M. Conte, North Carolina State University (Appendix E); José Duato, Universitat Politècnica de

València and Simula (Appendix F); David Goldberg, Xerox PARC (Appendix J); Timothy M. Pinkston, University of Southern California (Appendix F).

José Flich of the Universidad Politécnica de Valencia provided significant contributions to the updating of Appendix F.

Case Studies With Exercises

Jason D. Bakos, University of South Carolina (Chapters 3 and 4); Rajeev Balasubramonian, University of Utah (Chapter 2); Diana Franklin, The University of Chicago (Chapter 1 and Appendix C); Norman P. Jouppi, Google, (Chapter 2); Naveen Muralimanohar, HP Labs (Chapter 2); Gregory Peterson, University of Tennessee (Appendix A); Parthasarathy Ranganathan, Google (Chapter 6); Cliff Young, Google (Chapter 7); Amr Zaky, University of Santa Clara (Chapter 5 and Appendix B).

Jichuan Chang, Junwhan Ahn, Rama Govindaraju, and Milad Hashemi assisted in the development and testing of the case studies and exercises for Chapter 6.

Additional Material

John Nickolls, Steve Keckler, and Michael Toksvig of NVIDIA (Chapter 4 NVIDIA GPUs); Victor Lee, Intel (Chapter 4 comparison of Core i7 and GPU); John Shalf, LBNL (Chapter 4 recent vector architectures); Sam Williams, LBNL (Roofline model for computers in Chapter 4); Steve Blackburn of Australian National University and Kathryn McKinley of University of Texas at Austin (Intel performance and power measurements in Chapter 5); Luiz Barroso, Urs Hölzle, Jimmy Clidaris, Bob Felderman, and Chris Johnson of Google (the Google WSC in Chapter 6); James Hamilton of Amazon Web Services (power distribution and cost model in Chapter 6).

Jason D. Bakos of the University of South Carolina updated the lecture slides for this edition.

This book could not have been published without a publisher, of course. We wish to thank all the Morgan Kaufmann/Elsevier staff for their efforts and support. For this fifth edition, we particularly want to thank our editors Nate McFadden and Steve Merken, who coordinated surveys, development of the case studies and exercises, manuscript reviews, and the updating of the appendices.

We must also thank our university staff, Margaret Rowland and Roxana Infante, for countless express mailings, as well as for holding down the fort at Stanford and Berkeley while we worked on the book.

Our final thanks go to our wives for their suffering through increasingly early mornings of reading, thinking, and writing.

Contributors to Previous Editions

Reviewers

George Adams, Purdue University; Sarita Adve, University of Illinois at Urbana-Champaign; Jim Archibald, Brigham Young University; Krste Asanović, Massachusetts Institute of Technology; Jean-Loup Baer, University of Washington; Paul Barr, Northeastern University; Rajendra V. Boppana, University of Texas, San Antonio; Mark Brehob, University of Michigan; Doug Burger, University of Texas, Austin; John Burger, SGI; Michael Butler; Thomas Casavant; Rohit Chandra; Peter Chen, University of Michigan; the classes at SUNY Stony Brook, Carnegie Mellon, Stanford, Clemson, and Wisconsin; Tim Coe, Vitesse Semiconductor; Robert P. Colwell; David Cummings; Bill Dally; David Douglas; José Duato, Universitat Politècnica de València and Simula; Anthony Duben, Southeast Missouri State University; Susan Eggers, University of Washington; Joel Emer; Barry Fagin, Dartmouth; Joel Ferguson, University of California, Santa Cruz; Carl Feynman; David Filo; Josh Fisher, Hewlett-Packard Laboratories; Rob Fowler, DIKU; Mark Franklin, Washington University (St. Louis); Kourosh Gharachorloo; Nikolas Gloy, Harvard University; David Goldberg, Xerox Palo Alto Research Center; Antonio González, Intel and Universitat Politècnica de Catalunya; James Goodman, University of Wisconsin-Madison; Sudhanva Gurumurthi, University of Virginia; David Harris, Harvey Mudd College; John Heinlein; Mark Heinrich, Stanford; Daniel Helman, University of California, Santa Cruz; Mark D. Hill, University of Wisconsin-Madison; Martin Hopkins, IBM; Jerry Huck, Hewlett-Packard Laboratories; Wen-mei Hwu, University of Illinois at Urbana-Champaign; Mary Jane Irwin, Pennsylvania State University; Truman Joe; Norm Jouppi; David Kaeli, Northeastern University; Roger Kieckhafer, University of Nebraska; Lev G. Kirischian, Ryerson University; Earl Killian; Allan Knies, Purdue University; Don Knuth; Jeff Kuskin, Stanford; James R. Larus, Microsoft Research; Corinna Lee, University of Toronto; Hank Levy; Kai Li, Princeton University; Lori Liebrock, University of Alaska, Fairbanks; Mikko Lipasti, University of Wisconsin-Madison; Gyula A. Mago, University of North Carolina, Chapel Hill; Bryan Martin; Norman Matloff; David Meyer; William Michalson, Worcester Polytechnic Institute; James Mooney; Trevor Mudge, University of Michigan; Ramadass Nagarajan, University of Texas at Austin; David Nagle, Carnegie Mellon University; Todd Narter; Victor Nelson; Vojin Oklobdzija, University of California, Berkeley; Kunle Olukotun, Stanford University; Bob Owens, Pennsylvania State University; Greg Papadapoulous, Sun Microsystems; Joseph Pfeiffer; Keshav Pingali, Cornell University; Timothy M. Pinkston, University of Southern California; Bruno Preiss, University of Waterloo; Steven Przybylski; Jim Quinlan; Andras Radics; Kishore Ramachandran, Georgia Institute of Technology; Joseph Rameh, University of Texas, Austin; Anthony Reeves, Cornell University; Richard Reid, Michigan State University; Steve Reinhardt, University of Michigan; David Rennels, University of California, Los Angeles; Arnold L. Rosenberg, University of Massachusetts, Amherst; Kaushik Roy, Purdue

University; Emilio Salgueiro, Unysis; Karthikeyan Sankaralingam, University of Texas at Austin; Peter Schnorf; Margo Seltzer; Behrooz Shirazi, Southern Methodist University; Daniel Siewiorek, Carnegie Mellon University; J. P. Singh, Princeton; Ashok Singhal; Jim Smith, University of Wisconsin-Madison; Mike Smith, Harvard University; Mark Smotherman, Clemson University; Gurindar Sohi, University of Wisconsin-Madison; Arun Somani, University of Washington; Gene Tagliarin, Clemson University; Shyamkumar Thoziyoor, University of Notre Dame; Evan Tick, University of Oregon; Akhilesh Tyagi, University of North Carolina, Chapel Hill; Dan Upton, University of Virginia; Mateo Valero, Universidad Politécnica de Cataluña, Barcelona; Anujan Varma, University of California, Santa Cruz; Thorsten von Eicken, Cornell University; Hank Walker, Texas A&M; Roy Want, Xerox Palo Alto Research Center; David Weaver, Sun Microsystems; Shlomo Weiss, Tel Aviv University; David Wells; Mike Westall, Clemson University; Maurice Wilkes; Eric Williams; Thomas Willis, Purdue University; Malcolm Wing; Larry Wittie, SUNY Stony Brook; Ellen Witte Zegura, Georgia Institute of Technology; Sotirios G. Ziavras, New Jersey Institute of Technology.

Appendices

The vector appendix was revised by Krste Asanović of the Massachusetts Institute of Technology. The floating-point appendix was written originally by David Goldberg of Xerox PARC.

Exercises

George Adams, Purdue University; Todd M. Bezenek, University of Wisconsin-Madison (in remembrance of his grandmother Ethel Eshom); Susan Eggers; Anoop Gupta; David Hayes; Mark Hill; Allan Knies; Ethan L. Miller, University of California, Santa Cruz; Parthasarathy Ranganathan, Compaq Western Research Laboratory; Brandon Schwartz, University of Wisconsin-Madison; Michael Scott; Dan Siewiorek; Mike Smith; Mark Smotherman; Evan Tick; Thomas Willis.

Case Studies With Exercises

Andrea C. Arpaci-Dusseau, University of Wisconsin-Madison; Remzi H. Arpaci-Dusseau, University of Wisconsin-Madison; Robert P. Colwell, R&E Colwell & Assoc., Inc.; Diana Franklin, California Polytechnic State University, San Luis Obispo; Wen-mei W. Hwu, University of Illinois at Urbana-Champaign; Norman P. Jouppi, HP Labs; John W. Sias, University of Illinois at Urbana-Champaign; David A. Wood, University of Wisconsin-Madison.

Special Thanks

Duane Adams, Defense Advanced Research Projects Agency; Tom Adams; Sarita Adve, University of Illinois at Urbana-Champaign; Anant Agarwal; Dave

Albonesi, University of Rochester; Mitch Alsup; Howard Alt; Dave Anderson; Peter Ashenden; David Bailey; Bill Bandy, Defense Advanced Research Projects Agency; Luiz Barroso, Compaq's Western Research Lab; Andy Bechtolsheim; *C. Gordon* Bell; Fred Berkowitz; John Best, IBM; Dileep Bhandarkar; Jeff Bier, BDTI; Mark Birman; David Black; David Boggs; Jim Brady; Forrest Brewer; Aaron Brown, University of California, Berkeley; E. Bugnion, Compaq's Western Research Lab; Alper Buyuktosunoglu, University of Rochester; Mark Callaghan; Jason F. Cantin; Paul Carrick; Chen-Chung Chang; Lei Chen, University of Rochester; Pete Chen; Nhan Chu; Doug Clark, Princeton University; Bob Cmelik; John Crawford; Zarka Cvetanovic; Mike Dahlin, University of Texas, Austin; Merrick Darley; the staff of the DEC Western Research Laboratory; John DeRosa; Lloyd Dickman; J. Ding; Susan Eggers, University of Washington; Wael El-Essawy, University of Rochester; Patty Enriquez, Mills; Milos Ercegovac; Robert Garner; K. Gharachorloo, Compaq's Western Research Lab; Garth Gibson; Ronald Greenberg; Ben Hao; John Henning, Compaq; Mark Hill, University of Wisconsin-Madison; Danny Hillis; David Hodges; Urs Hölzle, Google; David Hough; Ed Hudson; Chris Hughes, University of Illinois at Urbana-Champaign; Mark Johnson; Lewis Jordan; Norm Jouppi; William Kahan; Randy Katz; Ed Kelly; Richard Kessler; Les Kohn; John Kowaleski, Compaq Computer Corp; Dan Lambright; Gary Lauterbach, Sun Microsystems; Corinna Lee; Ruby Lee; Don Lewine; Chao-Huang Lin; Paul Losleben, Defense Advanced Research Projects Agency; Yung-Hsiang Lu; Bob Lucas, Defense Advanced Research Projects Agency; Ken Lutz; Alan Mainwaring, Intel Berkeley Research Labs; Al Marston; Rich Martin, Rutgers; John Mashey; Luke McDowell; Sebastian Mirolo, Trimedia Corporation; Ravi Murthy; Biswadeep Nag; Lisa Noordergraaf, Sun Microsystems; Bob Parker, Defense Advanced Research Projects Agency; Vern Paxson, Center for Internet Research; Lawrence Prince; Steven Przybylski; Mark Pullen, Defense Advanced Research Projects Agency; Chris Rowen; Margaret Rowland; Greg Semeraro, University of Rochester; Bill Shannon; Behrooz Shirazi; Robert Shomler; Jim Slager; Mark Smotherman, Clemson University; the SMT research group at the University of Washington; Steve Squires, Defense Advanced Research Projects Agency; Ajay Sreekanth; Darren Staples; Charles Stapper; Jorge Stolfi; Peter Stoll; the students at Stanford and Berkeley who endured our first attempts at creating this book; Bob Supnik; Steve Swanson; Paul Taysom; Shreekant Thakkar; Alexander Thomasian, New Jersey Institute of Technology; John Toole, Defense Advanced Research Projects Agency; Kees A. Vissers, Trimedia Corporation; Willa Walker; David Weaver; Ric Wheeler, EMC; Maurice Wilkes; Richard Zimmerman.

John Hennessy ■ *David Patterson*

1.1 Introduction 2
1.2 Classes of Computers 6
1.3 Defining Computer Architecture 11
1.4 Trends in Technology 18
1.5 Trends in Power and Energy in Integrated Circuits 23
1.6 Trends in Cost 29
1.7 Dependability 36
1.8 Measuring, Reporting, and Summarizing Performance 39
1.9 Quantitative Principles of Computer Design 48
1.10 Putting It All Together: Performance, Price, and Power 55
1.11 Fallacies and Pitfalls 58
1.12 Concluding Remarks 64
1.13 Historical Perspectives and References 67
Case Studies and Exercises by Diana Franklin 67

1

Fundamentals of Quantitative Design and Analysis

An iPod, a phone, an Internet mobile communicator… these are NOT three separate devices! And we are calling it iPhone! Today Apple is going to reinvent the phone. And here it is.

Steve Jobs, January 9, 2007

New information and communications technologies, in particular high-speed Internet, are changing the way companies do business, transforming public service delivery and democratizing innovation. With 10 percent increase in high speed Internet connections, economic growth increases by 1.3 percent.

The World Bank, July 28, 2009

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00001-8
© 2019 Elsevier Inc. All rights reserved.

1.1 Introduction

Computer technology has made incredible progress in the roughly 70 years since the first general-purpose electronic computer was created. Today, less than $500 will purchase a cell phone that has as much performance as the world's fastest computer bought in 1993 for $50 million. This rapid improvement has come both from advances in the technology used to build computers and from innovations in computer design.

Although technological improvements historically have been fairly steady, progress arising from better computer architectures has been much less consistent. During the first 25 years of electronic computers, both forces made a major contribution, delivering performance improvement of about 25% per year. The late 1970s saw the emergence of the microprocessor. The ability of the microprocessor to ride the improvements in integrated circuit technology led to a higher rate of performance improvement—roughly 35% growth per year.

This growth rate, combined with the cost advantages of a mass-produced microprocessor, led to an increasing fraction of the computer business being based on microprocessors. In addition, two significant changes in the computer marketplace made it easier than ever before to succeed commercially with a new architecture. First, the virtual elimination of assembly language programming reduced the need for object-code compatibility. Second, the creation of standardized, vendor-independent operating systems, such as UNIX and its clone, Linux, lowered the cost and risk of bringing out a new architecture.

These changes made it possible to develop successfully a new set of architectures with simpler instructions, called RISC (Reduced Instruction Set Computer) architectures, in the early 1980s. The RISC-based machines focused the attention of designers on two critical performance techniques, the exploitation of *instruction-level parallelism* (initially through pipelining and later through multiple instruction issue) and the use of caches (initially in simple forms and later using more sophisticated organizations and optimizations).

The RISC-based computers raised the performance bar, forcing prior architectures to keep up or disappear. The Digital Equipment Vax could not, and so it was replaced by a RISC architecture. Intel rose to the challenge, primarily by translating 80x86 instructions into RISC-like instructions internally, allowing it to adopt many of the innovations first pioneered in the RISC designs. As transistor counts soared in the late 1990s, the hardware overhead of translating the more complex x86 architecture became negligible. In low-end applications, such as cell phones, the cost in power and silicon area of the x86-translation overhead helped lead to a RISC architecture, ARM, becoming dominant.

Figure 1.1 shows that the combination of architectural and organizational enhancements led to 17 years of sustained growth in performance at an annual rate of over 50%—a rate that is unprecedented in the computer industry.

The effect of this dramatic growth rate during the 20th century was fourfold. First, it has significantly enhanced the capability available to computer users. For many applications, the highest-performance microprocessors outperformed the supercomputer of less than 20 years earlier.

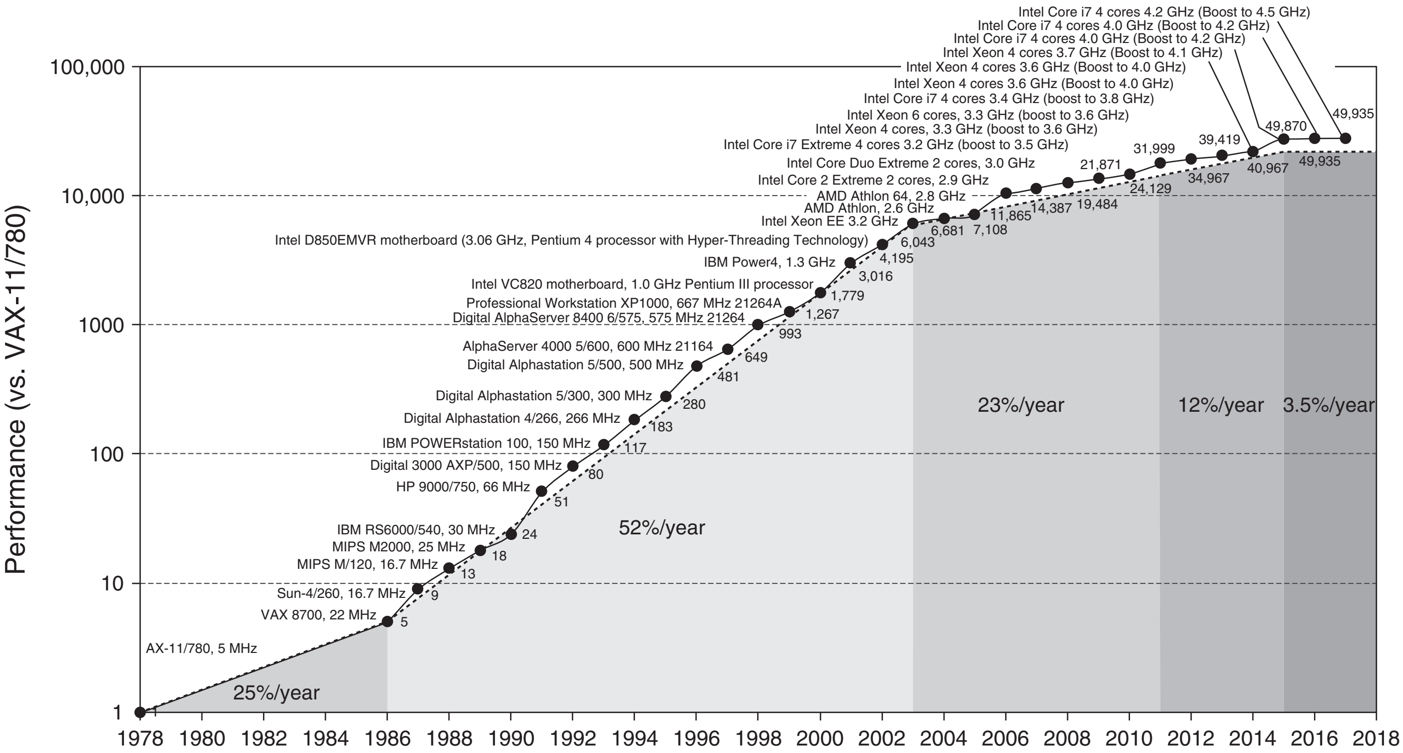


Figure 1.1 Growth in processor performance over 40 years. This chart plots program performance relative to the VAX 11/780 as measured by the SPEC integer benchmarks (see Section 1.8). Prior to the mid-1980s, growth in processor performance was largely technology-driven and averaged about 22% per year, or doubling performance every 3.5 years. The increase in growth to about 52% starting in 1986, or doubling every 2 years, is attributable to more advanced architectural and organizational ideas typified in RISC architectures. By 2003 this growth led to a difference in performance of an approximate factor of 25 versus the performance that would have occurred if it had continued at the 22% rate. In 2003 the limits of power due to the end of Dennard scaling and the available instruction-level parallelism slowed uniprocessor performance to 23% per year until 2011, or doubling every 3.5 years. (The fastest SPECintbase performance since 2007 has had automatic parallelization turned on, so uniprocessor speed is harder to gauge. These results are limited to single-chip systems with usually four cores per chip.) From 2011 to 2015, the annual improvement was less than 12%, or doubling every 8 years in part due to the limits of parallelism of Amdahl's Law. Since 2015, with the end of Moore's Law, improvement has been just 3.5% per year, or doubling every 20 years! Performance for floating-point-oriented calculations follows the same trends, but typically has 1% to 2% higher annual growth in each shaded region. Figure 1.11 on page 27 shows the improvement in clock rates for these same eras. Because SPEC has changed over the years, performance of newer machines is estimated by a scaling factor that relates the performance for different versions of SPEC: SPEC89, SPEC92, SPEC95, SPEC2000, and SPEC2006. There are too few results for SPEC2017 to plot yet.

Second, this dramatic improvement in cost-performance led to new classes of computers. Personal computers and workstations emerged in the 1980s with the availability of the microprocessor. The past decade saw the rise of smart cell phones and tablet computers, which many people are using as their primary computing platforms instead of PCs. These mobile client devices are increasingly using the Internet to access warehouses containing 100,000 servers, which are being designed as if they were a single gigantic computer.

Third, improvement of semiconductor manufacturing as predicted by Moore's law has led to the dominance of microprocessor-based computers across the entire range of computer design. Minicomputers, which were traditionally made from off-the-shelf logic or from gate arrays, were replaced by servers made by using microprocessors. Even mainframe computers and high-performance supercomputers are all collections of microprocessors.

The preceding hardware innovations led to a renaissance in computer design, which emphasized both architectural innovation and efficient use of technology improvements. This rate of growth compounded so that by 2003, high-performance microprocessors were 7.5 times as fast as what would have been obtained by relying solely on technology, including improved circuit design, that is, 52% per year versus 35% per year.

This hardware renaissance led to the fourth impact, which was on software development. This 50,000-fold performance improvement since 1978 (see Figure 1.1) allowed modern programmers to trade performance for productivity. In place of performance-oriented languages like C and C++, much more programming today is done in managed programming languages like Java and Scala. Moreover, scripting languages like JavaScript and Python, which are even more productive, are gaining in popularity along with programming frameworks like AngularJS and Django. To maintain productivity and try to close the performance gap, interpreters with just-in-time compilers and trace-based compiling are replacing the traditional compiler and linker of the past. Software deployment is changing as well, with Software as a Service (SaaS) used over the Internet replacing shrink-wrapped software that must be installed and run on a local computer.

The nature of applications is also changing. Speech, sound, images, and video are becoming increasingly important, along with predictable response time that is so critical to the user experience. An inspiring example is Google Translate. This application lets you hold up your cell phone to point its camera at an object, and the image is sent wirelessly over the Internet to a warehouse-scale computer (WSC) that recognizes the text in the photo and translates it into your native language. You can also speak into it, and it will translate what you said into audio output in another language. It translates text in 90 languages and voice in 15 languages.

Alas, Figure 1.1 also shows that this 17-year hardware renaissance is over. The fundamental reason is that two characteristics of semiconductor processes that were true for decades no longer hold.

In 1974 Robert Dennard observed that power density was constant for a given area of silicon even as you increased the number of transistors because of smaller dimensions of each transistor. Remarkably, transistors could go faster but use less power. *Dennard scaling* ended around 2004 because current and voltage couldn't keep dropping and still maintain the dependability of integrated circuits.

This change forced the microprocessor industry to use multiple efficient processors or cores instead of a single inefficient processor. Indeed, in 2004 Intel canceled its high-performance uniprocessor projects and joined others in declaring that the road to higher performance would be via multiple processors per chip rather than via faster uniprocessors. This milestone signaled a historic switch from relying solely on instruction-level parallelism (ILP), the primary focus of the first three editions of this book, to *data-level parallelism* (DLP) and *thread-level parallelism* (TLP), which were featured in the fourth edition and expanded in the fifth edition. The fifth edition also added WSCs and *request-level parallelism* (RLP), which is expanded in this edition. Whereas the compiler and hardware conspire to exploit ILP implicitly without the programmer's attention, DLP, TLP, and RLP are explicitly parallel, requiring the restructuring of the application so that it can exploit explicit parallelism. In some instances, this is easy; in many, it is a major new burden for programmers.

Amdahl's Law (Section 1.9) prescribes practical limits to the number of useful cores per chip. If 10% of the task is serial, then the maximum performance benefit from parallelism is 10 no matter how many cores you put on the chip.

The second observation that ended recently is *Moore's Law*. In 1965 Gordon Moore famously predicted that the number of transistors per chip would double every year, which was amended in 1975 to every two years. That prediction lasted for about 50 years, but no longer holds. For example, in the 2010 edition of this book, the most recent Intel microprocessor had 1,170,000,000 transistors. If Moore's Law had continued, we could have expected microprocessors in 2016 to have 18,720,000,000 transistors. Instead, the equivalent Intel microprocessor has just 1,750,000,000 transistors, or off by a factor of 10 from what Moore's Law would have predicted.

The combination of

- transistors no longer getting much better because of the slowing of Moore's Law and the end of Dinnard scaling,
- the unchanging power budgets for microprocessors,
- the replacement of the single power-hungry processor with several energy-efficient processors, and
- the limits to multiprocessing to achieve Amdahl's Law

caused improvements in processor performance to slow down, that is, to *double every 20 years,* rather than every 1.5 years as it did between 1986 and 2003 (see Figure 1.1).

The only path left to improve energy-performance-cost is specialization. Future microprocessors will include several domain-specific cores that perform only one class of computations well, but they do so remarkably better than general-purpose cores. The new Chapter 7 in this edition introduces *domain-specific architectures*.

This text is about the architectural ideas and accompanying compiler improvements that made the incredible growth rate possible over the past century, the reasons for the dramatic change, and the challenges and initial promising approaches to architectural ideas, compilers, and interpreters for the 21st century. At the core is a quantitative approach to computer design and analysis that uses empirical observations of programs, experimentation, and simulation as its tools. It is this style and approach to computer design that is reflected in this text. The purpose of this chapter is to lay the quantitative foundation on which the following chapters and appendices are based.

This book was written not only to explain this design style but also to stimulate you to contribute to this progress. We believe this approach will serve the computers of the future just as it worked for the implicitly parallel computers of the past.

1.2 Classes of Computers

These changes have set the stage for a dramatic change in how we view computing, computing applications, and the computer markets in this new century. Not since the creation of the personal computer have we seen such striking changes in the way computers appear and in how they are used. These changes in computer use have led to five diverse computing markets, each characterized by different applications, requirements, and computing technologies. Figure 1.2 summarizes these mainstream classes of computing environments and their important characteristics.

Internet of Things/Embedded Computers

Embedded computers are found in everyday machines: microwaves, washing machines, most printers, networking switches, and all automobiles. The phrase

Feature	Personal mobile device (PMD)	Desktop	Server	Clusters/warehouse-scale computer	Internet of things/ embedded
Price of system	\$100–\$1000	\$300–\$2500	\$5000–\$10,000,000	\$100,000–\$200,000,000	\$10–\$100,000
Price of microprocessor	\$10–\$100	\$50–\$500	\$200–\$2000	\$50–\$250	\$0.01–\$100
Critical system design issues	Cost, energy, media performance, responsiveness	Price-performance, energy, graphics performance	Throughput, availability, scalability, energy	Price-performance, throughput, energy proportionality	Price, energy, application-specific performance

Figure 1.2 A summary of the five mainstream computing classes and their system characteristics. Sales in 2015 included about 1.6 billion PMDs (90% cell phones), 275 million desktop PCs, and 15 million servers. The total number of embedded processors sold was nearly 19 billion. In total, 14.8 billion ARM-technology-based chips were shipped in 2015. Note the wide range in system price for servers and embedded systems, which go from USB keys to network routers. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing.

Internet of Things (IoT) refers to embedded computers that are connected to the Internet, typically wirelessly. When augmented with sensors and actuators, IoT devices collect useful data and interact with the physical world, leading to a wide variety of "smart" applications, such as smart watches, smart thermostats, smart speakers, smart cars, smart homes, smart grids, and smart cities.

Embedded computers have the widest spread of processing power and cost. They include 8-bit to 32-bit processors that may cost one penny, and high-end 64-bit processors for cars and network switches that cost $100. Although the range of computing power in the embedded computing market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal often meets the performance need at a minimum price, rather than achieving more performance at a higher price. The projections for the number of IoT devices in 2020 range from 20 to 50 billion.

Most of this book applies to the design, use, and performance of embedded processors, whether they are off-the-shelf microprocessors or microprocessor cores that will be assembled with other special-purpose hardware.

Unfortunately, the data that drive the quantitative design and evaluation of other classes of computers have not yet been extended successfully to embedded computing (see the challenges with EEMBC, for example, in Section 1.8). Hence we are left for now with qualitative descriptions, which do not fit well with the rest of the book. As a result, the embedded material is concentrated in Appendix E. We believe a separate appendix improves the flow of ideas in the text while allowing readers to see how the differing requirements affect embedded computing.

Personal Mobile Device

Personal mobile device (PMD) is the term we apply to a collection of wireless devices with multimedia user interfaces such as cell phones, tablet computers, and so on. Cost is a prime concern given the consumer price for the whole product is a few hundred dollars. Although the emphasis on energy efficiency is frequently driven by the use of batteries, the need to use less expensive packaging—plastic versus ceramic—and the absence of a fan for cooling also limit total power consumption. We examine the issue of energy and power in more detail in Section 1.5. Applications on PMDs are often web-based and media-oriented, like the previously mentioned Google Translate example. Energy and size requirements lead to use of Flash memory for storage (Chapter 2) instead of magnetic disks.

The processors in a PMD are often considered embedded computers, but we are keeping them as a separate category because PMDs are platforms that can run externally developed software, and they share many of the characteristics of desktop computers. Other embedded devices are more limited in hardware and software sophistication. We use the ability to run third-party software as the dividing line between nonembedded and embedded computers.

Responsiveness and predictability are key characteristics for media applications. A *real-time performance* requirement means a segment of the application has an absolute maximum execution time. For example, in playing a video on a

PMD, the time to process each video frame is limited, since the processor must accept and process the next frame shortly. In some applications, a more nuanced requirement exists: the average time for a particular task is constrained as well as the number of instances when some maximum time is exceeded. Such approaches—sometimes called *soft real-time*—arise when it is possible to miss the time constraint on an event occasionally, as long as not too many are missed. Real-time performance tends to be highly application-dependent.

Other key characteristics in many PMD applications are the need to minimize memory and the need to use energy efficiently. Energy efficiency is driven by both battery power and heat dissipation. The memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. The importance of memory size translates to an emphasis on code size, since data size is dictated by the application.

Desktop Computing

The first, and possibly still the largest market in dollar terms, is desktop computing. Desktop computing spans from low-end netbooks that sell for under $300 to high-end, heavily configured workstations that may sell for $2500. Since 2008, more than half of the desktop computers made each year have been battery operated laptop computers. Desktop computing sales are declining.

Throughout this range in price and capability, the desktop market tends to be driven to optimize *price-performance.* This combination of performance (measured primarily in terms of compute performance and graphics performance) and price of a system is what matters most to customers in this market, and hence to computer designers. As a result, the newest, highest-performance microprocessors and cost-reduced microprocessors often appear first in desktop systems (see Section 1.6 for a discussion of the issues affecting the cost of computers).

Desktop computing also tends to be reasonably well characterized in terms of applications and benchmarking, though the increasing use of web-centric, interactive applications poses new challenges in performance evaluation.

Servers

As the shift to desktop computing occurred in the 1980s, the role of servers grew to provide larger-scale and more reliable file and computing services. Such servers have become the backbone of large-scale enterprise computing, replacing the traditional mainframe.

For servers, different characteristics are important. First, availability is critical. (We discuss availability in Section 1.7.) Consider the servers running ATM machines for banks or airline reservation systems. Failure of such server systems is far more catastrophic than failure of a single desktop, since these servers must operate seven days a week, 24 hours a day. Figure 1.3 estimates revenue costs of downtime for server applications.

Application	Cost of downtime per hour	Annual losses with downtime of 1% (87.6 h/year)	0.5% (43.8 h/year)	0.1% (8.8 h/year)
Brokerage service	$4,000,000	$350,400,000	$175,200,000	$35,000,000
Energy	$1,750,000	$153,300,000	$76,700,000	$15,300,000
Telecom	$1,250,000	$109,500,000	$54,800,000	$11,000,000
Manufacturing	$1,000,000	$87,600,000	$43,800,000	$8,800,000
Retail	$650,000	$56,900,000	$28,500,000	$5,700,000
Health care	$400,000	$35,000,000	$17,500,000	$3,500,000
Media	$50,000	$4,400,000	$2,200,000	$400,000

Figure 1.3 Costs rounded to nearest $100,000 of an unavailable system are shown by analyzing the cost of downtime (in terms of immediately lost revenue), assuming three different levels of availability, and that downtime is distributed uniformly. These data are from Landstrom (2014) and were collected and analyzed by Contingency Planning Research.

A second key feature of server systems is scalability. Server systems often grow in response to an increasing demand for the services they support or an expansion in functional requirements. Thus the ability to scale up the computing capacity, the memory, the storage, and the I/O bandwidth of a server is crucial.

Finally, servers are designed for efficient throughput. That is, the overall performance of the server—in terms of transactions per minute or web pages served per second—is what is crucial. Responsiveness to an individual request remains important, but overall efficiency and cost-effectiveness, as determined by how many requests can be handled in a unit time, are the key metrics for most servers. We return to the issue of assessing performance for different types of computing environments in Section 1.8.

Clusters/Warehouse-Scale Computers

The growth of Software as a Service (SaaS) for applications like search, social networking, video viewing and sharing, multiplayer games, online shopping, and so on has led to the growth of a class of computers called *clusters*. Clusters are collections of desktop computers or servers connected by local area networks to act as a single larger computer. Each node runs its own operating system, and nodes communicate using a networking protocol. WSCs are the largest of the clusters, in that they are designed so that tens of thousands of servers can act as one. Chapter 6 describes this class of extremely large computers.

Price-performance and power are critical to WSCs since they are so large. As Chapter 6 explains, the majority of the cost of a warehouse is associated with power and cooling of the computers inside the warehouse. The annual amortized computers themselves and the networking gear cost for a WSC is $40 million, because they are usually replaced every few years. When you are buying that much computing, you need to buy wisely, because a 10% improvement in price-performance means an annual savings of $4 million (10% of $40 million) per WSC; a company like Amazon might have 100 WSCs!

WSCs are related to servers in that availability is critical. For example, Amazon.com had $136 billion in sales in 2016. As there are about 8800 hours in a year, the average revenue per hour was about $15 million. During a peak hour for Christmas shopping, the potential loss would be many times higher. As Chapter 6 explains, the difference between WSCs and servers is that WSCs use redundant, inexpensive components as the building blocks, relying on a software layer to catch and isolate the many failures that will happen with computing at this scale to deliver the availability needed for such applications. Note that scalability for a WSC is handled by the local area network connecting the computers and not by integrated computer hardware, as in the case of servers.

Supercomputers are related to WSCs in that they are equally expensive, costing hundreds of millions of dollars, but supercomputers differ by emphasizing floating-point performance and by running large, communication-intensive batch programs that can run for weeks at a time. In contrast, WSCs emphasize interactive applications, large-scale storage, dependability, and high Internet bandwidth.

Classes of Parallelism and Parallel Architectures

Parallelism at multiple levels is now the driving force of computer design across all four classes of computers, with energy and cost being the primary constraints. There are basically two kinds of parallelism in applications:

1. *Data-level parallelism (DLP)* arises because there are many data items that can be operated on at the same time.
2. *Task-level parallelism (TLP)* arises because tasks of work are created that can operate independently and largely in parallel.

Computer hardware in turn can exploit these two kinds of application parallelism in four major ways:

1. *Instruction-level parallelism* exploits data-level parallelism at modest levels with compiler help using ideas like pipelining and at medium levels using ideas like speculative execution.
2. *Vector architectures, graphic processor units (GPUs), and multimedia instruction sets* exploit data-level parallelism by applying a single instruction to a collection of data in parallel.
3. *Thread-level parallelism* exploits either data-level parallelism or task-level parallelism in a tightly coupled hardware model that allows for interaction between parallel threads.
4. *Request-level parallelism* exploits parallelism among largely decoupled tasks specified by the programmer or the operating system.

When Flynn (1966) studied the parallel computing efforts in the 1960s, he found a simple classification whose abbreviations we still use today. They target data-level parallelism and task-level parallelism. He looked at the parallelism in the instruction and data streams called for by the instructions at the most constrained component of the multiprocessor and placed all computers in one of four categories:

1. *Single instruction stream, single data stream* (SISD)—This category is the uniprocessor. The programmer thinks of it as the standard sequential computer, but it can exploit ILP. Chapter 3 covers SISD architectures that use ILP techniques such as superscalar and speculative execution.
2. *Single instruction stream, multiple data streams* (SIMD)—The same instruction is executed by multiple processors using different data streams. SIMD computers exploit *data-level parallelism* by applying the same operations to multiple items of data in parallel. Each processor has its own data memory (hence, the MD of SIMD), but there is a single instruction memory and control processor, which fetches and dispatches instructions. Chapter 4 covers DLP and three different architectures that exploit it: vector architectures, multimedia extensions to standard instruction sets, and GPUs.
3. *Multiple instruction streams, single data stream* (MISD)—No commercial multiprocessor of this type has been built to date, but it rounds out this simple classification.
4. *Multiple instruction streams, multiple data streams* (MIMD)—Each processor fetches its own instructions and operates on its own data, and it targets task-level parallelism. In general, MIMD is more flexible than SIMD and thus more generally applicable, but it is inherently more expensive than SIMD. For example, MIMD computers can also exploit data-level parallelism, although the overhead is likely to be higher than would be seen in an SIMD computer. This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. Chapter 5 covers tightly coupled MIMD architectures, which exploit *thread-level parallelism* because multiple cooperating threads operate in parallel. Chapter 6 covers loosely coupled MIMD architectures—specifically, *clusters* and *warehouse-scale computers*—that exploit *request-level parallelism*, where many independent tasks can proceed in parallel naturally with little need for communication or synchronization.

This taxonomy is a coarse model, as many parallel processors are hybrids of the SISD, SIMD, and MIMD classes. Nonetheless, it is useful to put a framework on the design space for the computers we will see in this book.

1.3 Defining Computer Architecture

The task the computer designer faces is a complex one: determine what attributes are important for a new computer, then design a computer to maximize performance and energy efficiency while staying within cost, power, and availability constraints. This task has many aspects, including instruction set design, functional organization, logic design, and implementation. The implementation may encompass integrated circuit design, packaging, power, and cooling. Optimizing the design requires familiarity with a very wide range of technologies, from compilers and operating systems to logic design and packaging.

A few decades ago, the term *computer architecture* generally referred to only instruction set design. Other aspects of computer design were called *implementation,* often insinuating that implementation is uninteresting or less challenging.

We believe this view is incorrect. The architect's or designer's job is much more than instruction set design, and the technical hurdles in the other aspects of the project are likely more challenging than those encountered in instruction set design. We'll quickly review instruction set architecture before describing the larger challenges for the computer architect.

Instruction Set Architecture: The Myopic View of Computer Architecture

We use the term *instruction set architecture* (ISA) to refer to the actual programmer-visible instruction set in this book. The ISA serves as the boundary between the software and hardware. This quick review of ISA will use examples from 80x86, ARMv8, and RISC-V to illustrate the seven dimensions of an ISA. The most popular RISC processors come from ARM (Advanced RISC Machine), which were in 14.8 billion chips shipped in 2015, or roughly 50 times as many chips that shipped with 80x86 processors. Appendices A and K give more details on the three ISAs.

RISC-V ("RISC Five") is a modern RISC instruction set developed at the University of California, Berkeley, which was made free and openly adoptable in response to requests from industry. In addition to a full software stack (compilers, operating systems, and simulators), there are several RISC-V implementations freely available for use in custom chips or in field-programmable gate arrays. Developed 30 years after the first RISC instruction sets, RISC-V inherits its ancestors' good ideas—a large set of registers, easy-to-pipeline instructions, and a lean set of operations—while avoiding their omissions or mistakes. It is a free and open, elegant example of the RISC architectures mentioned earlier, which is why more than 60 companies have joined the RISC-V foundation, including AMD, Google, HP Enterprise, IBM, Microsoft, Nvidia, Qualcomm, Samsung, and Western Digital. We use the integer core ISA of RISC-V as the example ISA in this book.

1. *Class of ISA*—Nearly all ISAs today are classified as general-purpose register architectures, where the operands are either registers or memory locations. The 80x86 has 16 general-purpose registers and 16 that can hold floating-point data, while RISC-V has 32 general-purpose and 32 floating-point registers (see Figure 1.4). The two popular versions of this class are *register-memory* ISAs,

Register	Name	Use	Saver
`x0`	`zero`	The constant value 0	N.A.
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	–
`x4`	`tp`	Thread pointer	–
`x5-x7`	`t0-t2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x11`	`a0-a1`	Function arguments/return values	Caller
`x12-x17`	`a2-a7`	Function arguments	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x28-x31`	`t3-t6`	Temporaries	Caller
`f0-f7`	`ft0-ft7`	FP temporaries	Caller
`f8-f9`	`fs0-fs1`	FP saved registers	Callee
`f10-f11`	`fa0-fa1`	FP function arguments/return values	Caller
`f12-f17`	`fa2-fa7`	FP function arguments	Caller
`f18-f27`	`fs2-fs11`	FP saved registers	Callee
`f28-f31`	`ft8-ft11`	FP temporaries	Caller

Figure 1.4 RISC-V registers, names, usage, and calling conventions. In addition to the 32 general-purpose registers (x0–x31), RISC-V has 32 floating-point registers (f0–f31) that can hold either a 32-bit single-precision number or a 64-bit double-precision number. The registers that are preserved across a procedure call are labeled "Callee" saved.

such as the 80x86, which can access memory as part of many instructions, and *load-store* ISAs, such as ARMv8 and RISC-V, which can access memory only with load or store instructions. All ISAs announced since 1985 are load-store.

2. *Memory addressing*—Virtually all desktop and server computers, including the 80x86, ARMv8, and RISC-V, use byte addressing to access memory operands. Some architectures, like ARMv8, require that objects must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. (See Figure A.5 on page A-8.) The 80x86 and RISC-V do not require alignment, but accesses are generally faster if operands are aligned.

3. *Addressing modes*—In addition to specifying registers and constant operands, addressing modes specify the address of a memory object. RISC-V addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three modes, plus three variations of displacement: no register (absolute), two registers (based indexed with displacement), and two registers

where one register is multiplied by the size of the operand in bytes (based with scaled index and displacement). It has more like the last three modes, minus the displacement field, plus register indirect, indexed, and based with scaled index. ARMv8 has the three RISC-V addressing modes plus PC-relative addressing, the sum of two registers, and the sum of two registers where one register is multiplied by the size of the operand in bytes. It also has autoincrement and autodecrement addressing, where the calculated address replaces the contents of one of the registers used in forming the address.

4. *Types and sizes of operands*—Like most ISAs, 80x86, ARMv8, and RISC-V support operand sizes of 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision). The 80x86 also supports 80-bit floating point (extended double precision).

5. *Operations*—The general categories of operations are data transfer, arithmetic logical, control (discussed next), and floating point. RISC-V is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures being used in 2017. Figure 1.5 summarizes the integer RISC-V ISA, and Figure 1.6 lists the floating-point ISA. The 80x86 has a much richer and larger set of operations (see Appendix K).

6. *Control flow instructions*—Virtually all ISAs, including these three, support conditional branches, unconditional jumps, procedure calls, and returns. All three use PC-relative addressing, where the branch address is specified by an address field that is added to the PC. There are some small differences. RISC-V conditional branches (`BE`, `BNE`, etc.) test the contents of registers, and the 80x86 and ARMv8 branches test condition code bits set as side effects of arithmetic/logic operations. The ARMv8 and RISC-V procedure call places the return address in a register, whereas the 80x86 call (`CALLF`) places the return address on a stack in memory.

7. *Encoding an ISA*—There are two basic choices on encoding: *fixed length* and *variable length*. All ARMv8 and RISC-V instructions are 32 bits long, which simplifies instruction decoding. Figure 1.7 shows the RISC-V instruction formats. The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable-length instructions can take less space than fixed-length instructions, so a program compiled for the 80x86 is usually smaller than the same program compiled for RISC-V. Note that choices mentioned previously will affect how the instructions are encoded into a binary representation. For example, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, because the register field and addressing mode field can appear many times in a single instruction. (Note that ARMv8 and RISC-V later offered extensions, called Thumb-2 and RV64IC, that provide a mix of 16-bit and 32-bit length instructions, respectively, to reduce program size. Code size for these compact versions of RISC architectures are smaller than that of the 80x86. See Appendix K.)

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 12-bit displacement + contents of a GPR*
`lb`, `lbu`, `sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh`, `lhu`, `sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw`, `lwu`, `sw`	Load word, load word unsigned, store word (to/from integer registers)
`ld`, `sd`	Load double word, store double word (to/from integer registers)
`flw`, `fld`, `fsw`, `fsd`	Load SP float, load DP float, store SP float, store DP float
`fmv._.x`, `fmv.x._`	Copy from/to integer register to/from floating-point register; "__" = S for single-precision, D for double-precision
`csrrw`, `csrrwi`, `csrrs`, `csrrsi`, `csrrc`, `csrrci`	Read counters and write status registers, which include counters: clock cycles, time, instructions retired
Arithmetic/logical	*Operations on integer or logical data in GPRs*
`add`, `addi`, `addw`, `addiw`	Add, add immediate (all immediates are 12 bits), add 32-bits only & sign-extend to 64 bits, add immediate 32-bits only
`sub`, `subw`	Subtract, subtract 32-bits only
`mul`, `mulw`, `mulh`, `mulhsu`, `mulhu`	Multiply, multiply 32-bits only, multiply upper half, multiply upper half signed-unsigned, multiply upper half unsigned
`div`, `divu`, `rem`, `remu`	Divide, divide unsigned, remainder, remainder unsigned
`divw`, `divuw`, `remw`, `remuw`	Divide and remainder: as previously, but divide only lower 32-bits, producing 32-bit sign-extended result
`and`, `andi`	And, and immediate
`or`, `ori`, `xor`, `xori`	Or, or immediate, exclusive or, exclusive or immediate
`lui`	Load upper immediate; loads bits 31-12 of register with immediate, then sign-extends
`auipc`	Adds immediate in bits 31–12 with zeros in lower bits to PC; used with `JALR` to transfer control to any 32-bit address
`sll`, `slli`, `srl`, `srli`, `sra`, `srai`	Shifts: shift left logical, right logical, right arithmetic; both variable and immediate forms
`sllw`, `slliw`, `srlw`, `srliw`, `sraw`, `sraiw`	Shifts: as previously, but shift lower 32-bits, producing 32-bit sign-extended result
`slt`, `slti`, `sltu`, `sltiu`	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`	Branch GPR equal/not equal; less than; greater than or equal, signed and unsigned
`jal`, `jalr`	Jump and link: save PC + 4, target is PC-relative (`JAL`) or a register (`JALR`); if specify `x0` as destination register, then acts as a simple jump
`ecall`	Make a request to the supporting execution environment, which is usually an OS
`ebreak`	Debuggers used to cause control to be transferred back to a debugging environment
`fence`, `fence.i`	Synchronize threads to guarantee ordering of memory accesses; synchronize instructions and data for stores to instruction memory

Figure 1.5 Subset of the instructions in RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions: multiply-divide (RVM), single-precision floating point (RVF), double-precision floating point (RVD). This figure includes RVM and the next one shows RVF and RVD. Appendix A gives much more detail on RISC-V.

Instruction type/opcode	**Instruction meaning**
Floating point	*FP operations on DP and SP formats*
`fadd.d`, `fadd.s`	Add DP, SP numbers
`fsub.d`, `fsub.s`	Subtract DP, SP numbers
`fmul.d`, `fmul.s`	Multiply DP, SP floating point
`fmadd.d`, `fmadd.s`, `fnmadd.d`, `fnmadd.s`	Multiply-add DP, SP numbers; negative multiply-add DP, SP numbers
`fmsub.d`, `fmsub.s`, `fnmsub.d`, `fnmsub.s`	Multiply-sub DP, SP numbers; negative multiply-sub DP, SP numbers
`fdiv.d`, `fdiv.s`	Divide DP, SP floating point
`fsqrt.d`, `fsqrt.s`	Square root DP, SP floating point
`fmax.d`, `fmax.s`, `fmin.d`, `fmin.s`	Maximum and minimum DP, SP floating point
`fcvt._._`, `fcvt._._u`, `fcvt._u._`	Convert instructions: `FCVT.x.y` converts from type `x` to type `y`, where `x` and `y` are `L` (64-bit integer), `W` (32-bit integer), `D` (DP), or `S` (SP). Integers can be unsigned (`U`)
`feq._`, `flt._`,`fle._`	Floating-point compare between floating-point registers and record the Boolean result in integer register; "__" = `S` for single-precision, `D` for double-precision
`fclass.d`, `fclass.s`	Writes to integer register a 10-bit mask that indicates the class of the floating-point number (−∞, +∞, −0, +0, NaN, …)
`fsgnj._`, `fsgnjn._`, `fsgnjx._`	Sign-injection instructions that changes only the sign bit: copy sign bit from other source, the oppositive of sign bit of other source, XOR of the 2 sign bits

Figure 1.6 Floating point instructions for RISC-V. RISC-V has a base set of instructions (R64I) and offers optional extensions for single-precision floating point (RVF) and double-precision floating point (RVD). SP = single precision; DP = double precision.

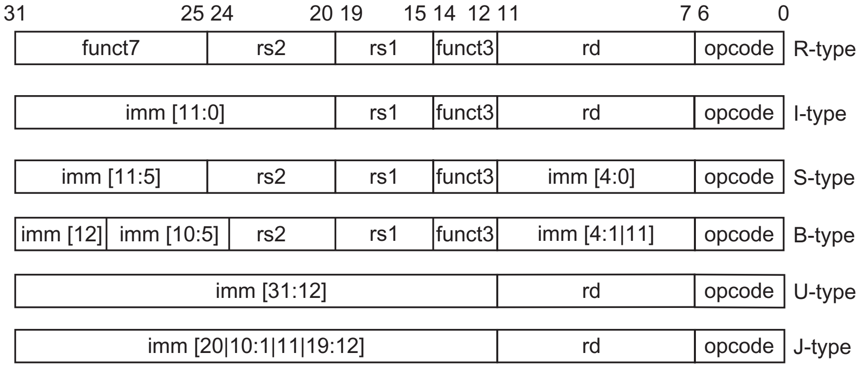

31
25 24
20 19
15 14
12 11
7 6
0
funct7
rs2
rs1
funct3
rd
opcode
R-type
imm [11:0]
rs1
funct3
rd
opcode
I-type
imm [11:5]
rs2
rs1
funct3
imm [4:0]
opcode
S-type
imm [12]
imm [10:5]
rs2
rs1
funct3
imm [4:1|11]
opcode
B-type
imm [31:12]
rd
opcode
U-type
imm [20|10:1|11|19:12]
rd
opcode
J-type

Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

The other challenges facing the computer architect beyond ISA design are particularly acute at the present, when the differences among instruction sets are small and when there are distinct application areas. Therefore, starting with the fourth edition of this book, beyond this quick review, the bulk of the instruction set material is found in the appendices (see Appendices A and K).

Genuine Computer Architecture: Designing the Organization and Hardware to Meet Goals and Functional Requirements

The implementation of a computer has two components: organization and hardware. The term *organization* includes the high-level aspects of a computer's design, such as the memory system, the memory interconnect, and the design of the internal processor or CPU (central processing unit—where arithmetic, logic, branching, and data transfer are implemented). The term *microarchitecture* is also used instead of organization. For example, two processors with the same instruction set architectures but different organizations are the AMD Opteron and the Intel Core i7. Both processors implement the 80x86 instruction set, but they have very different pipeline and cache organizations.

The switch to multiple processors per microprocessor led to the term *core* also being used for processors. Instead of saying multiprocessor microprocessor, the term *multicore* caught on. Given that virtually all chips have multiple processors, the term central processing unit, or CPU, is fading in popularity.

Hardware refers to the specifics of a computer, including the detailed logic design and the packaging technology of the computer. Often a line of computers contains computers with identical instruction set architectures and very similar organizations, but they differ in the detailed hardware implementation. For example, the Intel Core i7 (see Chapter 3) and the Intel Xeon E7 (see Chapter 5) are nearly identical but offer different clock rates and different memory systems, making the Xeon E7 more effective for server computers.

In this book, the word *architecture* covers all three aspects of computer design—instruction set architecture, organization or microarchitecture, and hardware.

Computer architects must design a computer to meet functional requirements as well as price, power, performance, and availability goals. Figure 1.8 summarizes requirements to consider in designing a new computer. Often, architects also must determine what the functional requirements are, which can be a major task. The requirements may be specific features inspired by the market. Application software typically drives the choice of certain functional requirements by determining how the computer will be used. If a large body of software exists for a particular instruction set architecture, the architect may decide that a new computer should implement an existing instruction set. The presence of a large market for a particular class of applications might encourage the designers to incorporate requirements that would make the computer competitive in that market. Later chapters examine many of these requirements and features in depth.

Functional requirements	**Typical features required or supported**
Application area	*Target of computer*
Personal mobile device	Real-time performance for a range of tasks, including interactive performance for graphics, video, and audio; energy efficiency (Chapters 2–5 and 7; Appendix A)
General-purpose desktop	Balanced performance for a range of tasks, including interactive performance for graphics, video, and audio (Chapters 2–5; Appendix A)
Servers	Support for databases and transaction processing; enhancements for reliability and availability; support for scalability (Chapters 2, 5, and 7; Appendices A, D, and F)
Clusters/warehouse-scale computers	Throughput performance for many independent tasks; error correction for memory; energy proportionality (Chapters 2, 6, and 7; Appendix F)
Internet of things/embedded computing	Often requires special support for graphics or video (or other application-specific extension); power limitations and power control may be required; real-time constraints (Chapters 2, 3, 5, and 7; Appendices A and E)
Level of software compatibility	*Determines amount of existing software for computer*
At programming language	Most flexible for designer; need new compiler (Chapters 3, 5, and 7; Appendix A)
Object code or binary compatible	Instruction set architecture is completely defined—little flexibility—but no investment needed in software or porting programs (Appendix A)
Operating system requirements	*Necessary features to support chosen OS* (Chapter 2; Appendix B)
Size of address space	Very important feature (Chapter 2); may limit applications
Memory management	Required for modern OS; may be paged or segmented (Chapter 2)
Protection	Different OS and application needs: page versus segment; virtual machines (Chapter 2)
Standards	*Certain standards may be required by marketplace*
Floating point	Format and arithmetic: IEEE 754 standard (Appendix J), special arithmetic for graphics or signal processing
I/O interfaces	For I/O devices: Serial ATA, Serial Attached SCSI, PCI Express (Appendices D and F)
Operating systems	UNIX, Windows, Linux, CISCO IOS
Networks	Support required for different networks: Ethernet, Infiniband (Appendix F)
Programming languages	Languages (ANSI C, C++, Java, Fortran) affect instruction set (Appendix A)

Figure 1.8 Summary of some of the most important functional requirements an architect faces. The left-hand column describes the class of requirement, while the right-hand column gives specific examples. The right-hand column also contains references to chapters and appendices that deal with the specific issues.

Architects must also be aware of important trends in both the technology and the use of computers because such trends affect not only the future cost but also the longevity of an architecture.

1.4 Trends in Technology

If an instruction set architecture is to prevail, it must be designed to survive rapid changes in computer technology. After all, a successful new instruction set architecture may last decades—for example, the core of the IBM mainframe has been in use for more than 50 years. An architect must plan for technology changes that can increase the lifetime of a successful computer.

To plan for the evolution of a computer, the designer must be aware of rapid changes in implementation technology. Five implementation technologies, which change at a dramatic pace, are critical to modern implementations:

- *Integrated circuit logic technology*—Historically, transistor density increased by about 35% per year, quadrupling somewhat over four years. Increases in die size are less predictable and slower, ranging from 10% to 20% per year. The combined effect was a traditional growth rate in transistor count on a chip of about 40%–55% per year, or doubling every 18–24 months. This trend is popularly known as Moore's Law. Device speed scales more slowly, as we discuss below. Shockingly, Moore's Law is no more. The number of devices per chip is still increasing, but at a decelerating rate. Unlike in the Moore's Law era, we expect the doubling time to be stretched with each new technology generation.
- *Semiconductor DRAM* (dynamic random-access memory)—This technology is the foundation of main memory, and we discuss it in Chapter 2. The growth of DRAM has slowed dramatically, from quadrupling every three years as in the past. The 8-gigabit DRAM was shipping in 2014, but the 16-gigabit DRAM won't reach that state until 2019, and it looks like there will be no 32-gigabit DRAM (Kim, 2005). Chapter 2 mentions several other technologies that may replace DRAM when it hits its capacity wall.
- *Semiconductor Flash* (electrically erasable programmable read-only memory)—This nonvolatile semiconductor memory is the standard storage device in PMDs, and its rapidly increasing popularity has fueled its rapid growth rate in capacity. In recent years, the capacity per Flash chip increased by about 50%–60% per year, doubling roughly every 2 years. Currently, Flash memory is 8–10 times cheaper per bit than DRAM. Chapter 2 describes Flash memory.
- *Magnetic disk technology*—Prior to 1990, density increased by about 30% per year, doubling in three years. It rose to 60% per year thereafter, and increased to 100% per year in 1996. Between 2004 and 2011, it dropped back to about 40% per year, or doubled every two years. Recently, disk improvement has slowed to less than 5% per year. One way to increase disk capacity is to add more platters at the same areal density, but there are already seven platters within the one-inch depth of the 3.5-inch form factor disks. There is room for at most one or two more platters. The last hope for real density increase is to use a small laser on each disk read-write head to heat a 30 nm spot to 400°C so that it can be written magnetically before it cools. It is unclear whether Heat Assisted Magnetic Recording can be manufactured economically and reliably, although Seagate announced plans to ship HAMR in limited production in 2018. HAMR is the last chance for continued improvement in areal density of hard disk

drives, which are now 8–10 times cheaper per bit than Flash and 200–300 times cheaper per bit than DRAM. This technology is central to server- and warehouse-scale storage, and we discuss the trends in detail in Appendix D.

- *Network technology*—Network performance depends both on the performance of switches and on the performance of the transmission system. We discuss the trends in networking in Appendix F.

These rapidly changing technologies shape the design of a computer that, with speed and technology enhancements, may have a lifetime of 3–5 years. Key technologies such as Flash change sufficiently that the designer must plan for these changes. Indeed, designers often design for the next technology, knowing that, when a product begins shipping in volume, the following technology may be the most cost-effective or may have performance advantages. Traditionally, cost has decreased at about the rate at which density increases.

Although technology improves continuously, the impact of these increases can be in discrete leaps, as a threshold that allows a new capability is reached. For example, when MOS technology reached a point in the early 1980s where between 25,000 and 50,000 transistors could fit on a single chip, it became possible to build a single-chip, 32-bit microprocessor. By the late 1980s, first-level caches could go on a chip. By eliminating chip crossings within the processor and between the processor and the cache, a dramatic improvement in cost-performance and energy-performance was possible. This design was simply unfeasible until the technology reached a certain point. With multicore microprocessors and increasing numbers of cores each generation, even server computers are increasingly headed toward a single chip for all processors. Such technology thresholds are not rare and have a significant impact on a wide variety of design decisions.

Performance Trends: Bandwidth Over Latency

As we shall see in Section 1.8, *bandwidth* or *throughput* is the total amount of work done in a given time, such as megabytes per second for a disk transfer. In contrast, *latency* or *response time* is the time between the start and the completion of an event, such as milliseconds for a disk access. Figure 1.9 plots the relative improvement in bandwidth and latency for technology milestones for microprocessors, memory, networks, and disks. Figure 1.10 describes the examples and milestones in more detail.

Performance is the primary differentiator for microprocessors and networks, so they have seen the greatest gains: 32,000–40,000 × in bandwidth and 50–90 × in latency. Capacity is generally more important than performance for memory and disks, so capacity has improved more, yet bandwidth advances of 400–2400 × are still much greater than gains in latency of 8–9 ×.

Clearly, bandwidth has outpaced latency across these technologies and will likely continue to do so. A simple rule of thumb is that bandwidth grows by at least the square of the improvement in latency. Computer designers should plan accordingly.

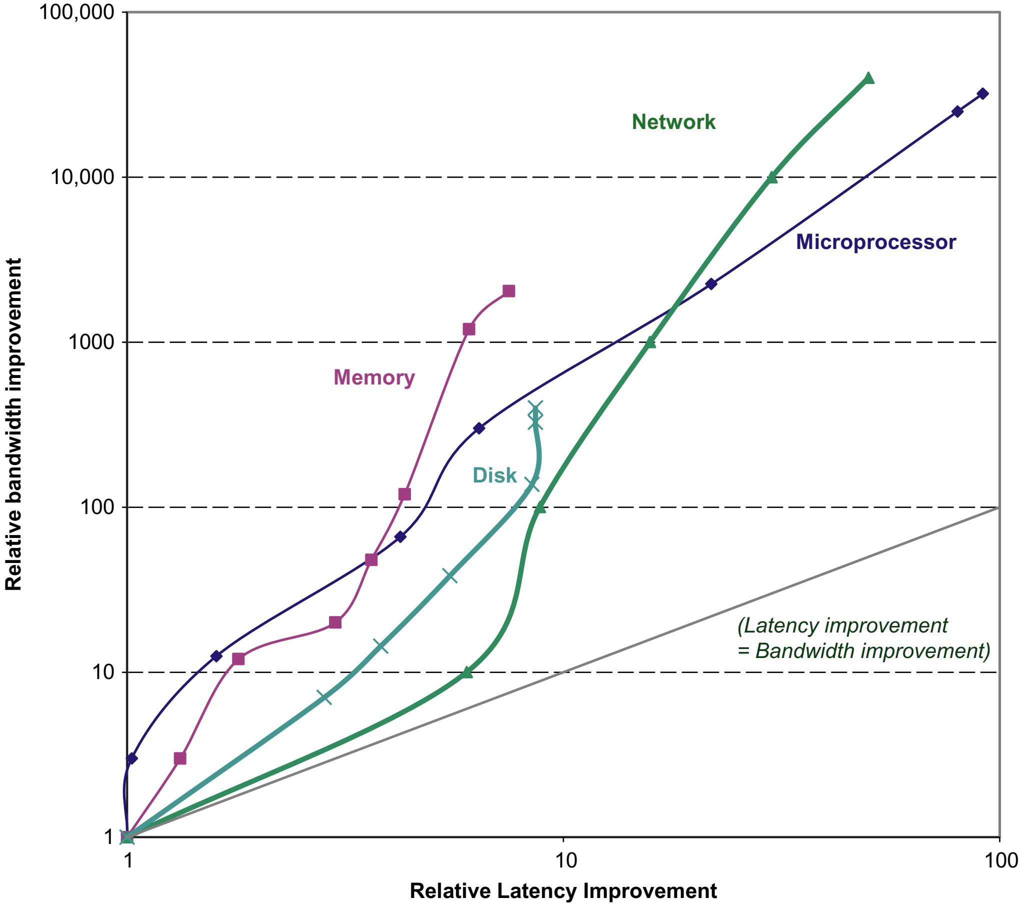


Figure 1.9 Log-log plot of bandwidth and latency milestones in Figure 1.10 relative to the first milestone. Note that latency improved 8–91 ×, while bandwidth improved about 400–32,000 ×. Except for networking, we note that there were modest improvements in latency and bandwidth in the other three technologies in the six years since the last edition: 0%–23% in latency and 23%–70% in bandwidth. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71–75.

Scaling of Transistor Performance and Wires

Integrated circuit processes are characterized by the *feature size*, which is the minimum size of a transistor or a wire in either the *x* or *y* dimension. Feature sizes decreased from 10 μm in 1971 to 0.016 μm in 2017; in fact, we have switched units, so production in 2017 is referred to as "16 nm," and 7 nm chips are underway. Since the transistor count per square millimeter of silicon is determined by the surface area of a transistor, the density of transistors increases quadratically with a linear decrease in feature size.

Microprocessor	16-Bit address/bus, microcoded	32-Bit address/bus, microcoded	5-Stage pipeline, on-chip I & D caches, FPU	2-Way superscalar, 64-bit bus	Out-of-order 3-way superscalar	Out-of-order superpipelined, on-chip L2 cache	Multicore OOO 4-way on chip L3 cache, Turbo
Product	Intel 80286	Intel 80386	Intel 80486	Intel Pentium	Intel Pentium Pro	Intel Pentium 4	Intel Core i7
Year	1982	1985	1989	1993	1997	2001	2015
Die size (mm^2)	47	43	81	90	308	217	122
Transistors	134,000	275,000	1,200,000	3,100,000	5,500,000	42,000,000	1,750,000,000
Processors/chip	1	1	1	1	1	1	4
Pins	68	132	168	273	387	423	1400
Latency (clocks)	6	5	5	5	10	22	14
Bus width (bits)	16	32	32	64	64	64	196
Clock rate (MHz)	12.5	16	25	66	200	1500	4000
Bandwidth (MIPS)	2	6	25	132	600	4500	64,000
Latency (ns)	320	313	200	76	50	15	4
Memory module	DRAM	Page mode DRAM	Fast page mode DRAM	Fast page mode DRAM	Synchronous DRAM	Double data rate SDRAM	DDR4 SDRAM
Module width (bits)	16	16	32	64	64	64	64
Year	1980	1983	1986	1993	1997	2000	2016
Mbits/DRAM chip	0.06	0.25	1	16	64	256	4096
Die size (mm^2)	35	45	70	130	170	204	50
Pins/DRAM chip	16	16	18	20	54	66	134
Bandwidth (MBytes/s)	13	40	160	267	640	1600	27,000
Latency (ns)	225	170	125	75	62	52	30
Local area network	Ethernet	Fast Ethernet	Gigabit Ethernet	10 Gigabit Ethernet	100 Gigabit Ethernet	400 Gigabit Ethernet	
IEEE standard	802.3	803.3u	802.3ab	802.3ac	802.3ba	802.3bs	
Year	1978	1995	1999	2003	2010	2017	
Bandwidth (Mbits/seconds)	10	100	1000	10,000	100,000	400,000	
Latency (μs)	3000	500	340	190	100	60	
Hard disk	3600 RPM	5400 RPM	7200 RPM	10,000 RPM	15,000 RPM	15,000 RPM	
Product	CDC WrenI 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453	Seagate ST600MX0062	
Year	1983	1990	1994	1998	2003	2016	
Capacity (GB)	0.03	1.4	4.3	9.1	73.4	600	
Disk form factor	5.25 in.	5.25 in.	3.5 in.	3.5 in.	3.5 in.	3.5 in.	
Media diameter	5.25 in.	5.25 in.	3.5 in.	3.0 in.	2.5 in.	2.5 in.	
Interface	ST-412	SCSI	SCSI	SCSI	SCSI	SAS	
Bandwidth (MBytes/s)	0.6	4	9	24	86	250	
Latency (ms)	48.3	17.1	12.7	8.8	5.7	3.6	

Figure 1.10 Performance milestones over 25–40 years for microprocessors, memory, networks, and disks. The microprocessor milestones are several generations of IA-32 processors, going from a 16-bit bus, microcoded 80286 to a 64-bit bus, multicore, out-of-order execution, superpipelined Core i7. Memory module milestones go from 16-bit-wide, plain DRAM to 64-bit-wide double data rate version 3 synchronous DRAM. Ethernet advanced from 10 Mbits/s to 400 Gbits/s. Disk milestones are based on rotation speed, improving from 3600 to 15,000 RPM. Each case is best-case bandwidth, and latency is the time for a simple operation assuming no contention. Updated from Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71–75.

The increase in transistor performance, however, is more complex. As feature sizes shrink, devices shrink quadratically in the horizontal dimension and also shrink in the vertical dimension. The shrink in the vertical dimension requires a reduction in operating voltage to maintain correct operation and reliability of the transistors. This combination of scaling factors leads to a complex interrelationship between transistor performance and process feature size. To a first approximation, in the past the transistor performance improved linearly with decreasing feature size.

The fact that transistor count improves quadratically with a linear increase in transistor performance is both the challenge and the opportunity for which computer architects were created! In the early days of microprocessors, the higher rate of improvement in density was used to move quickly from 4-bit, to 8-bit, to 16-bit, to 32-bit, to 64-bit microprocessors. More recently, density improvements have supported the introduction of multiple processors per chip, wider SIMD units, and many of the innovations in speculative execution and caches found in Chapters 2–5.

Although transistors generally improve in performance with decreased feature size, wires in an integrated circuit do not. In particular, the signal delay for a wire increases in proportion to the product of its resistance and capacitance. Of course, as feature size shrinks, wires get shorter, but the resistance and capacitance per unit length get worse. This relationship is complex, since both resistance and capacitance depend on detailed aspects of the process, the geometry of a wire, the loading on a wire, and even the adjacency to other structures. There are occasional process enhancements, such as the introduction of copper, which provide one-time improvements in wire delay.

In general, however, wire delay scales poorly compared to transistor performance, creating additional challenges for the designer. In addition to the power dissipation limit, wire delay has become a major design obstacle for large integrated circuits and is often more critical than transistor switching delay. Larger and larger fractions of the clock cycle have been consumed by the propagation delay of signals on wires, but power now plays an even greater role than wire delay.

1.5 Trends in Power and Energy in Integrated Circuits

Today, energy is the biggest challenge facing the computer designer for nearly every class of computer. First, power must be brought in and distributed around the chip, and modern microprocessors use hundreds of pins and multiple interconnect layers just for power and ground. Second, power is dissipated as heat and must be removed.

Power and Energy: A Systems Perspective

How should a system architect or a user think about performance, power, and energy? From the viewpoint of a system designer, there are three primary concerns.

First, what is the maximum power a processor ever requires? Meeting this demand can be important to ensuring correct operation. For example, if a processor attempts to draw more power than a power-supply system can provide (by drawing more current than the system can supply), the result is typically a voltage drop, which can cause devices to malfunction. Modern processors can vary widely in power consumption with high peak currents; hence they provide voltage indexing methods that allow the processor to slow down and regulate voltage within a wider margin. Obviously, doing so decreases performance.

Second, what is the sustained power consumption? This metric is widely called the *thermal design power* (TDP) because it determines the cooling requirement. TDP is neither peak power, which is often 1.5 times higher, nor is it the actual average power that will be consumed during a given computation, which is likely to be lower still. A typical power supply for a system is typically sized to exceed the TDP, and a cooling system is usually designed to match or exceed TDP. Failure to provide adequate cooling will allow the junction temperature in the processor to exceed its maximum value, resulting in device failure and possibly permanent damage. Modern processors provide two features to assist in managing heat, since the highest power (and hence heat and temperature rise) can exceed the long-term average specified by the TDP. First, as the thermal temperature approaches the junction temperature limit, circuitry lowers the clock rate, thereby reducing power. Should this technique not be successful, a second thermal overload trap is activated to power down the chip.

The third factor that designers and users need to consider is energy and energy efficiency. Recall that power is simply energy per unit time: 1 watt = 1 joule per second. Which metric is the right one for comparing processors: energy or power? In general, energy is always a better metric because it is tied to a specific task and the time required for that task. In particular, the energy to complete a workload is equal to the average power times the execution time for the workload.

Thus, if we want to know which of two processors is more efficient for a given task, we need to compare energy consumption (not power) for executing the task. For example, processor A may have a 20% higher average power consumption than processor B, but if A executes the task in only 70% of the time needed by B, its energy consumption will be $1.2 \times 0.7 = 0.84$, which is clearly better.

One might argue that in a large server or cloud, it is sufficient to consider the average power, since the workload is often assumed to be infinite, but this is misleading. If our cloud were populated with processor Bs rather than As, then the cloud would do less work for the same amount of energy expended. Using energy to compare the alternatives avoids this pitfall. Whenever we have a fixed workload, whether for a warehouse-size cloud or a smartphone, comparing energy will be the right way to compare computer alternatives, because the electricity bill for the cloud and the battery lifetime for the smartphone are both determined by the energy consumed.

When is power consumption a useful measure? The primary legitimate use is as a constraint: for example, an air-cooled chip might be limited to 100 W. It can be used as a metric if the workload is fixed, but then it's just a variation of the true metric of energy per task.

Energy and Power Within a Microprocessor

For CMOS chips, the traditional primary energy consumption has been in switching transistors, also called *dynamic energy.* The energy required per transistor is proportional to the product of the capacitive load driven by the transistor and the square of the voltage:

$$\text{Energy}_{\text{dynamic}} \propto \text{Capacitive load} \times \text{Voltage}^2$$

This equation is the energy of pulse of the logic transition of $0 \rightarrow 1 \rightarrow 0$ or $1 \rightarrow 0 \rightarrow 1$. The energy of a single transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) is then:

$$\text{Energy}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$$

The power required per transistor is just the product of the energy of a transition multiplied by the frequency of transitions:

$$\text{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

For a fixed task, slowing clock rate reduces power, but not energy.

Clearly, dynamic power and energy are greatly reduced by lowering the voltage, so voltages have dropped from 5 V to just under 1 V in 20 years. The capacitive load is a function of the number of transistors connected to an output and the technology, which determines the capacitance of the wires and the transistors.

Example Some microprocessors today are designed to have adjustable voltage, so a 15% reduction in voltage may result in a 15% reduction in frequency. What would be the impact on dynamic energy and on dynamic power?

Answer Because the capacitance is unchanged, the answer for energy is the ratio of the voltages

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(\text{Voltage} \times 0.85)^2}{\text{Voltage}^2} = 0.85^2 = 0.72$$

which reduces energy to about 72% of the original. For power, we add the ratio of the frequencies

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = 0.72 \times \frac{(\text{Frequency switched} \times 0.85)}{\text{Frequency switched}} = 0.61$$

shrinking power to about 61% of the original.

As we move from one process to the next, the increase in the number of transistors switching and the frequency with which they change dominate the decrease in load capacitance and voltage, leading to an overall growth in power consumption and energy. The first microprocessors consumed less than a watt, and the first

32-bit microprocessors (such as the Intel 80386) used about 2 W, whereas a 4.0 GHz Intel Core i7-6700K consumes 95 W. Given that this heat must be dissipated from a chip that is about 1.5 cm on a side, we are near the limit of what can be cooled by air, and this is where we have been stuck for nearly a decade.

Given the preceding equation, you would expect clock frequency growth to slow down if we can't reduce voltage or increase power per chip. Figure 1.11 shows that this has indeed been the case since 2003, even for the microprocessors in Figure 1.1 that were the highest performers each year. Note that this period of flatter clock rates corresponds to the period of slow performance improvement range in Figure 1.1.

Distributing the power, removing the heat, and preventing hot spots have become increasingly difficult challenges. Energy is now the major constraint to using transistors; in the past, it was the raw silicon area. Therefore modern

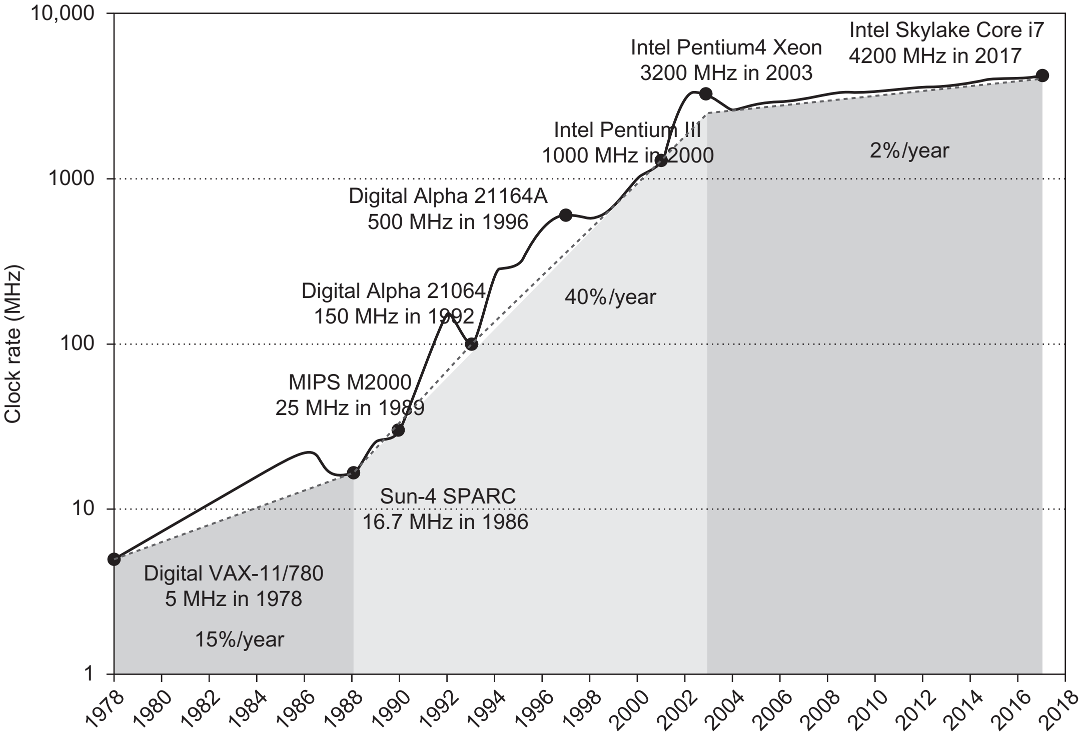


Figure 1.11 Growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 22% per year. During the "renaissance period" of 52% performance improvement per year between 1986 and 2003, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 2% per year, while single processor performance improved recently at just 3.5% per year.

microprocessors offer many techniques to try to improve energy efficiency despite flat clock rates and constant supply voltages:

1. *Do nothing well.* Most microprocessors today turn off the clock of inactive modules to save energy and dynamic power. For example, if no floating-point instructions are executing, the clock of the floating-point unit is disabled. If some cores are idle, their clocks are stopped.
2. *Dynamic voltage-frequency scaling (DVFS).* The second technique comes directly from the preceding formulas. PMDs, laptops, and even servers have periods of low activity where there is no need to operate at the highest clock frequency and voltages. Modern microprocessors typically offer a few clock frequencies and voltages in which to operate that use lower power and energy. Figure 1.12 plots the potential power savings via DVFS for a server as the workload shrinks for three different clock rates: 2.4, 1.8, and 1 GHz. The overall server power savings is about 10%–15% for each of the two steps.
3. *Design for the typical case.* Given that PMDs and laptops are often idle, memory and storage offer low power modes to save energy. For example, DRAMs have a series of increasingly lower power modes to extend battery life in PMDs and laptops, and there have been proposals for disks that have a mode that spins more slowly when unused to save power. However, you cannot access DRAMs or disks in these modes, so you must return to fully active mode to read or write, no matter how low the access rate. As mentioned, microprocessors for PCs have been designed instead for heavy use at high operating temperatures, relying on on-chip temperature sensors to detect when activity should be reduced automatically to avoid overheating. This "emergency slowdown" allows manufacturers to design for a more typical case and then rely on this safety mechanism if someone really does run programs that consume much more power than is typical.

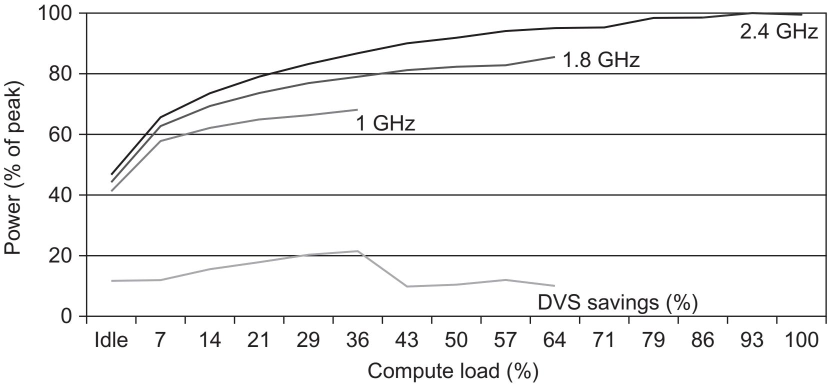


Figure 1.12 Energy savings for a server using an AMD Opteron microprocessor, 8 GB of DRAM, and one ATA disk. At 1.8 GHz, the server can handle at most up to two-thirds of the workload without causing service-level violations, and at 1 GHz, it can safely handle only one-third of the workload (Figure 5.11 in Barroso and Hölzle, 2009).

4. *Overclocking*. Intel started offering *Turbo mode* in 2008, where the chip decides that it is safe to run at a higher clock rate for a short time, possibly on just a few cores, until temperature starts to rise. For example, the 3.3 GHz Core i7 can run in short bursts for 3.6 GHz. Indeed, the highest-performing microprocessors each year since 2008 shown in Figure 1.1 have all offered temporary overclocking of about 10% over the nominal clock rate. For single-threaded code, these microprocessors can turn off all cores but one and run it faster. Note that, although the operating system can turn off Turbo mode, there is no notification once it is enabled, so the programmers may be surprised to see their programs vary in performance because of room temperature!

Although dynamic power is traditionally thought of as the primary source of power dissipation in CMOS, static power is becoming an important issue because leakage current flows even when a transistor is off:

$$\text{Power}_{\text{static}} \propto \text{Current}_{\text{static}} \times \text{Voltage}$$

That is, static power is proportional to the number of devices.

Thus increasing the number of transistors increases power even if they are idle, and current leakage increases in processors with smaller transistor sizes. As a result, very low-power systems are even turning off the power supply (*power gating*) to inactive modules in order to control loss because of leakage. In 2011 the goal for leakage was 25% of the total power consumption, with leakage in high-performance designs sometimes far exceeding that goal. Leakage can be as high as 50% for such chips, in part because of the large SRAM caches that need power to maintain the storage values. (The S in SRAM is for static.) The only hope to stop leakage is to turn off power to the chips' subsets.

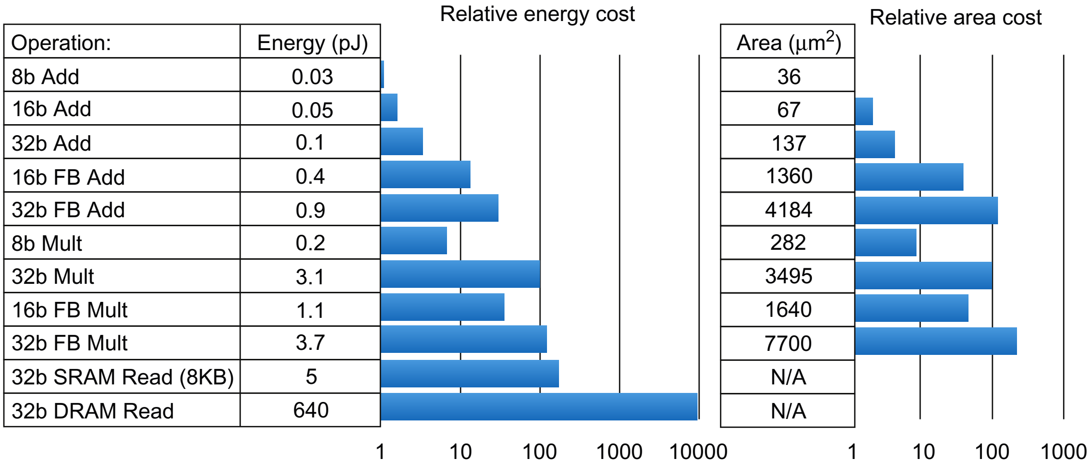
Finally, because the processor is just a portion of the whole energy cost of a system, it can make sense to use a faster, less energy-efficient processor to allow the rest of the system to go into a sleep mode. This strategy is known as *race-to-halt*.

The importance of power and energy has increased the scrutiny on the efficiency of an innovation, so the primary evaluation now is tasks per joule or performance per watt, contrary to performance per mm^2 of silicon as in the past. This new metric affects approaches to parallelism, as we will see in Chapters 4 and 5.

The Shift in Computer Architecture Because of Limits of Energy

As transistor improvement decelerates, computer architects must look elsewhere for improved energy efficiency. Indeed, given the energy budget, it is easy today to design a microprocessor with so many transistors that they cannot all be turned on at the same time. This phenomenon has been called *dark silicon*, in that much of a chip cannot be unused ("dark") at any moment in time because of thermal constraints. This observation has led architects to reexamine the fundamentals of processors' design in the search for a greater energy-cost performance.

Figure 1.13, which lists the energy cost and area cost of the building blocks of a modern computer, reveals surprisingly large ratios. For example, a 32-bit



Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FB Add	0.4	1360
32b FB Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FB Mult	1.1	1640
32b FB Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Energy numbers are from Mark Horowitz *Computing's Energy problem (and what we can do about it)*. ISSCC 2014
Area numbers are from synthesized result using Design compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Figure 1.13 Comparison of the energy and die area of arithmetic operations and energy cost of accesses to SRAM and DRAM. [Azizi][Dally]. Area is for TSMC 45 nm technology node.

floating-point addition uses 30 times as much energy as an 8-bit integer add. The area difference is even larger, by 60 times. However, the biggest difference is in memory; a 32-bit DRAM access takes 20,000 times as much energy as an 8-bit addition. A small SRAM is 125 times more energy-efficient than DRAM, which demonstrates the importance of careful uses of caches and memory buffers.

The new design principle of minimizing energy per task combined with the relative energy and area costs in Figure 1.13 have inspired a new direction for computer architecture, which we describe in Chapter 7. Domain-specific processors save energy by reducing wide floating-point operations and deploying special-purpose memories to reduce accesses to DRAM. They use those saving to provide 10–100 more (narrower) integer arithmetic units than a traditional processor. Although such processors perform only a limited set of tasks, they perform them remarkably faster and more energy efficiently than a general-purpose processor.

Like a hospital with general practitioners and medical specialists, computers in this energy-aware world will likely be combinations of general-purpose cores that can perform any task and special-purpose cores that do a few things extremely well and even more cheaply.

1.6 Trends in Cost

Although costs tend to be less important in some computer designs—specifically supercomputers—cost-sensitive designs are of growing significance. Indeed, in the past 35 years, the use of technology improvements to lower cost, as well as increase performance, has been a major theme in the computer industry.

Textbooks often ignore the cost half of cost-performance because costs change, thereby dating books, and because the issues are subtle and differ across industry segments. Nevertheless, it's essential for computer architects to have an understanding of cost and its factors in order to make intelligent decisions about whether a new feature should be included in designs where cost is an issue. (Imagine architects designing skyscrapers without any information on costs of steel beams and concrete!)

This section discusses the major factors that influence the cost of a computer and how these factors are changing over time.

The Impact of Time, Volume, and Commoditization

The cost of a manufactured computer component decreases over time even without significant improvements in the basic implementation technology. The underlying principle that drives costs down is the *learning curve*—manufacturing costs decrease over time. The learning curve itself is best measured by change in *yield*—the percentage of manufactured devices that survives the testing procedure. Whether it is a chip, a board, or a system, designs that have twice the yield will have half the cost.

Understanding how the learning curve improves yield is critical to projecting costs over a product's life. One example is that the price per megabyte of DRAM has dropped over the long term. Since DRAMs tend to be priced in close relationship to cost—except for periods when there is a shortage or an oversupply—price and cost of DRAM track closely.

Microprocessor prices also drop over time, but because they are less standardized than DRAMs, the relationship between price and cost is more complex. In a period of significant competition, price tends to track cost closely, although microprocessor vendors probably rarely sell at a loss.

Volume is a second key factor in determining cost. Increasing volumes affect cost in several ways. First, they decrease the time needed to get through the learning curve, which is partly proportional to the number of systems (or chips) manufactured. Second, volume decreases cost because it increases purchasing and manufacturing efficiency. As a rule of thumb, some designers have estimated that costs decrease about 10% for each doubling of volume. Moreover, volume decreases the amount of development costs that must be amortized by each computer, thus allowing cost and selling price to be closer and still make a profit.

Commodities are products that are sold by multiple vendors in large volumes and are essentially identical. Virtually all the products sold on the shelves of grocery stores are commodities, as are standard DRAMs, Flash memory, monitors, and keyboards. In the past 30 years, much of the personal computer industry has become a commodity business focused on building desktop and laptop computers running Microsoft Windows.

Because many vendors ship virtually identical products, the market is highly competitive. Of course, this competition decreases the gap between cost and selling price, but it also decreases cost. Reductions occur because a commodity market has both volume and a clear product definition, which allows multiple suppliers to compete in building components for the commodity product. As a result, the overall product cost is lower because of the competition among the suppliers of the components and the volume efficiencies the suppliers can achieve. This rivalry has led to the low end of the computer business being able to achieve better price-performance than other sectors and has yielded greater growth at the low end, although with very limited profits (as is typical in any commodity business).

Cost of an Integrated Circuit

Why would a computer architecture book have a section on integrated circuit costs? In an increasingly competitive computer marketplace where standard parts—disks, Flash memory, DRAMs, and so on—are becoming a significant portion of any system's cost, integrated circuit costs are becoming a greater portion of the cost that varies between computers, especially in the high-volume, cost-sensitive portion of the market. Indeed, with PMDs' increasing reliance of whole *systems on a chip* (SOC), the cost of the integrated circuits is much of the cost of the PMD. Thus computer designers must understand the costs of chips in order to understand the costs of current computers.

Although the costs of integrated circuits have dropped exponentially, the basic process of silicon manufacture is unchanged: A *wafer* is still tested and chopped into *dies* that are packaged (see Figures 1.14–1.16). Therefore the cost of a packaged integrated circuit is

$$\text{Cost of integrated circuit} = \frac{\text{Cost of die} + \text{Cost of testing die} + \text{Cost of packaging and final test}}{\text{Final test yield}}$$

In this section, we focus on the cost of dies, summarizing the key issues in testing and packaging at the end.

Learning how to predict the number of good chips per wafer requires first learning how many dies fit on a wafer and then learning how to predict the percentage of those that will work. From there it is simple to predict cost:

$$\text{Cost of die} = \frac{\text{Cost of wafer}}{\text{Dies per wafer} \times \text{Die yield}}$$

The most interesting feature of this initial term of the chip cost equation is its sensitivity to die size, shown below.

The number of dies per wafer is approximately the area of the wafer divided by the area of the die. It can be more accurately estimated by

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2 \times \text{Die area}}}$$

The first term is the ratio of wafer area (πr^2) to die area. The second compensates for the "square peg in a round hole" problem—rectangular dies near the periphery

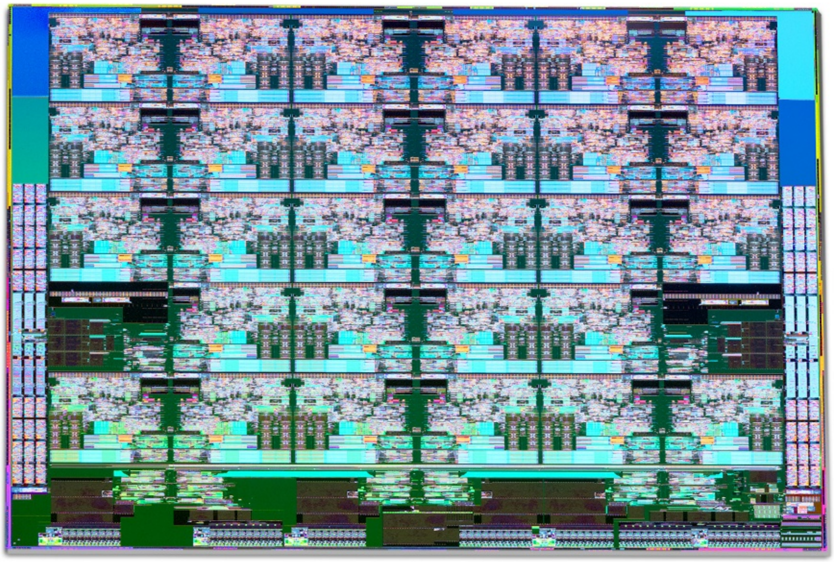


Figure 1.14 Photograph of an Intel Skylake microprocessor die, which is evaluated in Chapter 4.

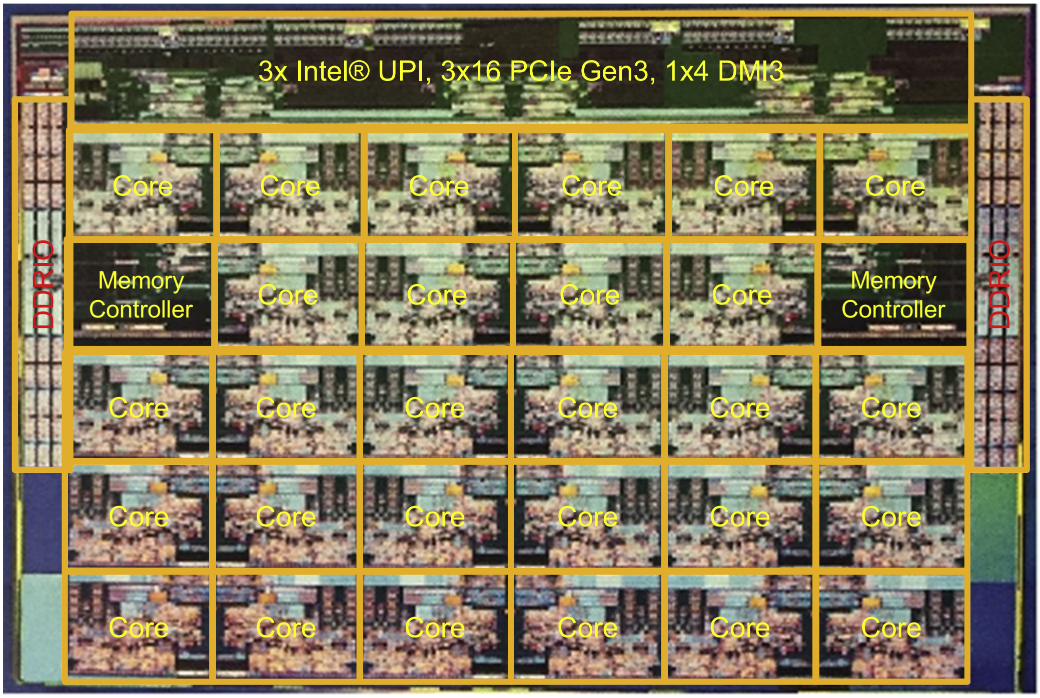

3x Intel® UPI, 3x16 PCIe Gen3, 1x4 DMI3
Core
Core
Core
Core
Core
Core
DDRIO
Memory Controller
Core
Core
Core
Core
Memory Controller
DDRIO
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core
Core

Figure 1.15 The components of the microprocessor die in Figure 1.14 are labeled with their functions.

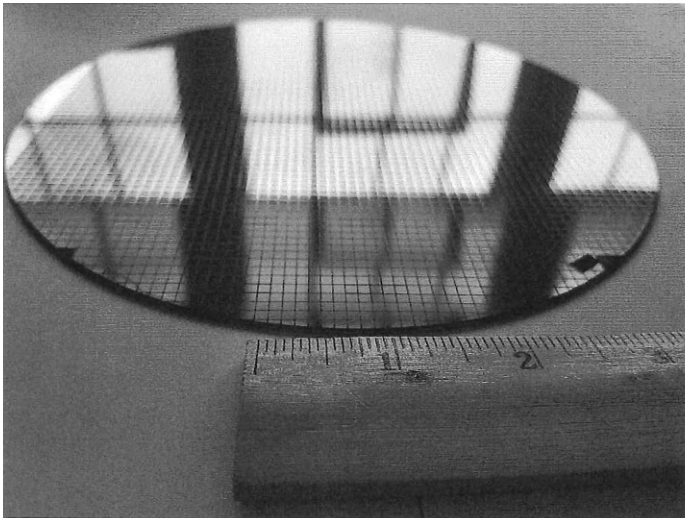


Figure 1.16 This 200 mm diameter wafer of RISC-V dies was designed by SiFive. It has two types of RISC-V dies using an older, larger processing line. An FE310 die is 2.65 mm × 2.72 mm and an SiFive test die that is 2.89 mm × 2.72 mm. The wafer contains 1846 of the former and 1866 of the latter, totaling 3712 chips.

of round wafers. Dividing the circumference (πd) by the diagonal of a square die is approximately the number of dies along the edge.

Example Find the number of dies per 300 mm (30 cm) wafer for a die that is 1.5 cm on a side and for a die that is 1.0 cm on a side.

Answer When die area is 2.25 cm^2:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = \frac{706.9}{2.25} - \frac{94.2}{2.12} = 270$$

Because the area of the larger die is 2.25 times bigger, there are roughly 2.25 as many smaller dies per wafer:

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{1.00} - \frac{\pi \times 30}{\sqrt{2 \times 1.00}} = \frac{706.9}{1.00} - \frac{94.2}{1.41} = 640$$

However, this formula gives only the maximum number of dies per wafer. The critical question is: What is the fraction of *good* dies on a wafer, or the *die yield*? A simple model of integrated circuit yield, which assumes that defects are randomly distributed over the wafer and that yield is inversely proportional to the complexity of the fabrication process, leads to the following:

$$\text{Die yield} = \text{Wafer yield} \times 1/(1 + \text{Defects per unit area} \times \text{Die area})^{N}$$

This Bose-Einstein formula is an empirical model developed by looking at the yield of many manufacturing lines (Sydow, 2006), and it still applies today. *Wafer yield* accounts for wafers that are completely bad and so need not be tested. For simplicity, we'll just assume the wafer yield is 100%. Defects per unit area is a measure of the random manufacturing defects that occur. In 2017 the value was typically 0.08–0.10 defects per square inch for a 28-nm node and 0.10–0.30 for the newer 16 nm node because it depends on the maturity of the process (recall the learning curve mentioned earlier). The metric versions are 0.012–0.016 defects per square centimeter for 28 nm and 0.016–0.047 for 16 nm. Finally, N is a parameter called the process-complexity factor, a measure of manufacturing difficulty. For 28 nm processes in 2017, N is 7.5–9.5. For a 16 nm process, N ranges from 10 to 14.

Example Find the die yield for dies that are 1.5 cm on a side and 1.0 cm on a side, assuming a defect density of 0.047 per cm^2 and N is 12.

Answer The total die areas are 2.25 and 1.00 cm^2. For the larger die, the yield is

$$\text{Die yield} = 1/(1 + 0.047 \times 2.25)^{12} \times 270 = 120$$

For the smaller die, the yield is

$$\text{Die yield} = 1/(1 + 0.047 \times 1.00)^{12} \times 640 = 444$$

The bottom line is the number of good dies per wafer. Less than half of all the large dies are good, but nearly 70% of the small dies are good.

Although many microprocessors fall between 1.00 and 2.25 cm^2, low-end embedded 32-bit processors are sometimes as small as 0.05 cm^2, processors used for embedded control (for inexpensive IoT devices) are often less than 0.01 cm^2, and high-end server and GPU chips can be as large as 8 cm^2.

Given the tremendous price pressures on commodity products such as DRAM and SRAM, designers have included redundancy as a way to raise yield. For a number of years, DRAMs have regularly included some redundant memory cells so that a certain number of flaws can be accommodated. Designers have used similar techniques in both standard SRAMs and in large SRAM arrays used for caches within microprocessors. GPUs have 4 redundant processors out of 84 for the same reason. Obviously, the presence of redundant entries can be used to boost the yield significantly.

In 2017 processing of a 300 mm (12-inch) diameter wafer in a 28-nm technology costs between \$4000 and \$5000, and a 16-nm wafer costs about \$7000. Assuming a processed wafer cost of \$7000, the cost of the 1.00 cm^2 die would be around \$16, but the cost per die of the 2.25 cm^2 die would be about \$58, or almost four times the cost of a die that is a little over twice as large.

What should a computer designer remember about chip costs? The manufacturing process dictates the wafer cost, wafer yield, and defects per unit area, so the sole control of the designer is die area. In practice, because the number of defects per unit area is small, the number of good dies per wafer, and therefore the cost per die, grows roughly as the square of the die area. The computer designer affects die size, and thus cost, both by what functions are included on or excluded from the die and by the number of I/O pins.

Before we have a part that is ready for use in a computer, the die must be tested (to separate the good dies from the bad), packaged, and tested again after packaging. These steps all add significant costs, increasing the total by half.

The preceding analysis focused on the variable costs of producing a functional die, which is appropriate for high-volume integrated circuits. There is, however, one very important part of the fixed costs that can significantly affect the cost of an integrated circuit for low volumes (less than 1 million parts), namely, the cost of a mask set. Each step in the integrated circuit process requires a separate mask. Therefore, for modern high-density fabrication processes with up to 10 metal layers, mask costs are about \$4 million for 16 nm and \$1.5 million for 28 nm.

The good news is that semiconductor companies offer “shuttle runs” to dramatically lower the costs of tiny test chips. They lower costs by putting many small designs onto a single die to amortize the mask costs, and then later split the dies into smaller pieces for each project. Thus TSMC delivers 80–100 untested dies that are 1.57 × 1.57 mm in a 28 nm process for \$30,000 in 2017. Although these die are tiny, they offer the architect millions of transistors to play with. For example, several RISC-V processors would fit on such a die.

Although shuttle runs help with prototyping and debugging runs, they don’t address small-volume production of tens to hundreds of thousands of parts. Because mask costs are likely to continue to increase, some designers are incorporating reconfigurable logic to enhance the flexibility of a part and thus reduce the cost implications of masks.

Cost Versus Price

With the commoditization of computers, the margin between the cost to manufacture a product and the price the product sells for has been shrinking. Those margins pay for a company’s research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes. Many engineers are surprised to find that most companies spend only 4% (in the commodity PC business) to 12% (in the high-end server business) of their income on R&D, which includes all engineering.

Cost of Manufacturing Versus Cost of Operation

For the first four editions of this book, cost meant the cost to build a computer and price meant price to purchase a computer. With the advent of WSCs, which contain tens of thousands of servers, the cost to operate the computers is significant in addition to the cost of purchase. Economists refer to these two costs as capital expenses (CAPEX) and operational expenses (OPEX).

As Chapter 6 shows, the amortized purchase price of servers and networks is about half of the monthly cost to operate a WSC, assuming a short lifetime of the IT equipment of 3–4 years. About 40% of the monthly operational costs are for power use and the amortized infrastructure to distribute power and to cool the IT equipment, despite this infrastructure being amortized over 10–15 years. Thus, to lower operational costs in a WSC, computer architects need to use energy efficiently.

1.7 Dependability

Historically, integrated circuits were one of the most reliable components of a computer. Although their pins may be vulnerable, and faults may occur over communication channels, the failure rate inside the chip was very low. That conventional wisdom is changing as we head to feature sizes of 16 nm and smaller, because both transient faults and permanent faults are becoming more commonplace, so architects must design systems to cope with these challenges. This section gives a quick overview of the issues in dependability, leaving the official definition of the terms and approaches to Section D.3 in Appendix D.

Computers are designed and constructed at different layers of abstraction. We can descend recursively down through a computer seeing components enlarge themselves to full subsystems until we run into individual transistors. Although some faults are widespread, like the loss of power, many can be limited to a single component in a module. Thus utter failure of a module at one level may be considered merely a component error in a higher-level module. This distinction is helpful in trying to find ways to build dependable computers.

One difficult question is deciding when a system is operating properly. This theoretical point became concrete with the popularity of Internet services. Infrastructure providers started offering *service level agreements* (SLAs) or *service level objectives* (SLOs) to guarantee that their networking or power service would be dependable. For example, they would pay the customer a penalty if they did not meet an agreement of some hours per month. Thus an SLA could be used to decide whether the system was up or down.

Systems alternate between two states of service with respect to an SLA:

1. *Service accomplishment*, where the service is delivered as specified.
2. *Service interruption*, where the delivered service is different from the SLA.

Transitions between these two states are caused by *failures* (from state 1 to state 2) or *restorations* (2 to 1). Quantifying these transitions leads to the two main measures of dependability:

- *Module reliability* is a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant. Therefore the *mean time to failure* (MTTF) is a reliability measure. The reciprocal of MTTF is a rate of failures, generally reported as failures per billion hours of operation, or *FIT* (for *failures in time*). Thus an MTTF of 1,000,000 hours equals $10^9/10^6$ or 1000 FIT. Service interruption is measured as *mean time to repair* (MTTR). *Mean time between failures* (MTBF) is simply the sum of MTTF + MTTR. Although MTBF is widely used, MTTF is often the more appropriate term. If a collection of modules has exponentially distributed lifetimes—meaning that the age of a module is not important in probability of failure—the overall failure rate of the collection is the sum of the failure rates of the modules.
- *Module availability* is a measure of the service accomplishment with respect to the alternation between the two states of accomplishment and interruption. For nonredundant systems with repair, module availability is

$$\text{Module availability} = \frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Note that reliability and availability are now quantifiable metrics, rather than synonyms for dependability. From these definitions, we can estimate reliability of a system quantitatively if we make some assumptions about the reliability of components and that failures are independent.

Example Assume a disk subsystem with the following components and MTTF:

- 10 disks, each rated at 1,000,000-hour MTTF
- 1 ATA controller, 500,000-hour MTTF
- 1 power supply, 200,000-hour MTTF
- 1 fan, 200,000-hour MTTF
- 1 ATA cable, 1,000,000-hour MTTF

Using the simplifying assumptions that the lifetimes are exponentially distributed and that failures are independent, compute the MTTF of the system as a whole.

Answer The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10+2+5+5+1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

The primary way to cope with failure is redundancy, either in time (repeat the operation to see if it still is erroneous) or in resources (have other components to take over from the one that failed). Once the component is replaced and the system is fully repaired, the dependability of the system is assumed to be as good as new. Let's quantify the benefits of redundancy with an example.

Example Disk subsystems often have redundant power supplies to improve dependability. Using the preceding components and MTTFs, calculate the reliability of redundant power supplies. Assume that one power supply is sufficient to run the disk subsystem and that we are adding one redundant power supply.

Answer We need a formula to show what to expect when we can tolerate a failure and still provide service. To simplify the calculations, we assume that the lifetimes of the components are exponentially distributed and that there is no dependency between the component failures. MTTF for our redundant power supplies is the mean time until one power supply fails divided by the chance that the other will fail before the first one is replaced. Thus, if the chance of a second failure before repair is small, then the MTTF of the pair is large.

Since we have two power supplies and independent failures, the mean time until one supply fails is $\text{MTTF}_{\text{power supply}}/2$. A good approximation of the probability of a second failure is MTTR over the mean time until the other power supply fails. Therefore a reasonable approximation for a redundant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}_{\text{power supply}}/2}{\dfrac{\text{MTTR}_{\text{power supply}}}{\text{MTTF}_{\text{power supply}}}} = \frac{\text{MTTF}^2_{\text{power supply}}/2}{\text{MTTR}_{\text{power supply}}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}}$$

Using the preceding MTTF numbers, if we assume it takes on average 24 hours for a human operator to notice that a power supply has failed and to replace it, the reliability of the fault tolerant pair of power supplies is

$$\text{MTTF}_{\text{power supply pair}} = \frac{\text{MTTF}^2_{\text{power supply}}}{2 \times \text{MTTR}_{\text{power supply}}} = \frac{200{,}000^2}{2 \times 24} \cong 830{,}000{,}000$$

making the pair about 4150 times more reliable than a single power supply.

Having quantified the cost, power, and dependability of computer technology, we are ready to quantify performance.

1.8 Measuring, Reporting, and Summarizing Performance

When we say one computer is faster than another one is, what do we mean? The user of a cell phone may say a computer is faster when a program runs in less time, while an Amazon.com administrator may say a computer is faster when it completes more transactions per hour. The cell phone user wants to reduce *response time*—the time between the start and the completion of an event—also referred to as *execution time*. The operator of a WSC wants to increase *throughput*—the total amount of work done in a given time.

In comparing design alternatives, we often want to relate the performance of two different computers, say, X and Y. The phrase "X is faster than Y" is used here to mean that the response time or execution time is lower on X than on Y for the given task. In particular, "X is n times as fast as Y" will mean

$$\frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = n$$

Since execution time is the reciprocal of performance, the following relationship holds:

$$n = \frac{\text{Execution time}_{\text{Y}}}{\text{Execution time}_{\text{X}}} = \frac{\dfrac{1}{\text{Performance}_{\text{Y}}}}{\dfrac{1}{\text{Performance}_{\text{X}}}} = \frac{\text{Performance}_{\text{X}}}{\text{Performance}_{\text{Y}}}$$

The phrase "the throughput of X is 1.3 times as fast as Y" signifies here that the number of tasks completed per unit time on computer X is 1.3 times the number completed on Y.

Unfortunately, time is not always the metric quoted in comparing the performance of computers. Our position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design.

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*, which is the latency to complete a task, including storage accesses, memory accesses, input/output activities, operating system overhead—everything. With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. Thus we need a term to consider this activity. *CPU time* recognizes this distinction and means the time the processor is computing, *not* including the time waiting for I/O or running other programs. (Clearly, the response time seen by the user is the elapsed time of the program, not the CPU time.)

Computer users who routinely run the same programs would be the perfect candidates to evaluate a new computer. To evaluate a new system, these users would simply compare the execution time of their *workloads*—the mixture of programs and operating system commands that users run on a computer. Few are in this happy situation, however. Most must rely on other methods to evaluate computers, and often other evaluators, hoping that these methods will predict performance for their usage of the new computer. One approach is benchmark programs, which are programs that many companies use to establish the relative performance of their computers.

Benchmarks

The best choice of benchmarks to measure performance is real applications, such as Google Translate mentioned in Section 1.1. Attempts at running programs that are much simpler than a real application have led to performance pitfalls. Examples include

- *Kernels*, which are small, key pieces of real applications.
- *Toy programs*, which are 100-line programs from beginning programming assignments, such as Quicksort.
- *Synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications, such as Dhrystone.

All three are discredited today, usually because the compiler writer and architect can conspire to make the computer appear faster on these stand-in programs than on real applications. Regrettably for your authors—who dropped the fallacy about using synthetic benchmarks to characterize performance in the fourth edition of this book since we thought all computer architects agreed it was disreputable—the synthetic program Dhrystone is still the most widely quoted benchmark for embedded processors in 2017!

Another issue is the conditions under which the benchmarks are run. One way to improve the performance of a benchmark has been with benchmark-specific compiler flags; these flags often caused transformations that would be illegal on many programs or would slow down performance on others. To restrict this process and increase the significance of the results, benchmark developers typically require the vendor to use one compiler and one set of flags for all the programs in the same language (such as C++ or C). In addition to the question of compiler flags, another question is whether source code modifications are allowed. There are three different approaches to addressing this question:

1. No source code modifications are allowed.
2. Source code modifications are allowed but are essentially impossible. For example, database benchmarks rely on standard database programs that are tens of millions of lines of code. The database companies are highly unlikely to make changes to enhance the performance for one particular computer.
3. Source modifications are allowed, as long as the altered version produces the same output.

The key issue that benchmark designers face in deciding to allow modification of the source is whether such modifications will reflect real practice and provide useful insight to users, or whether these changes simply reduce the accuracy of the benchmarks as predictors of real performance. As we will see in Chapter 7, domain-specific architects often follow the third option when creating processors for well-defined tasks.

To overcome the danger of placing too many eggs in one basket, collections of benchmark applications, called *benchmark suites*, are a popular measure of performance of processors with a variety of applications. Of course, such collections are only as good as the constituent individual benchmarks. Nonetheless, a key advantage of such suites is that the weakness of any one benchmark is lessened by the presence of the other benchmarks. The goal of a benchmark suite is that it will characterize the real relative performance of two computers, particularly for programs not in the suite that customers are likely to run.

A cautionary example is the Electronic Design News Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced “embassy”) benchmarks.

It is a set of 41 kernels used to predict performance of different embedded applications: automotive/industrial, consumer, networking, office automation, and telecommunications. EEMBC reports unmodified performance and “full fury” performance, where almost anything goes. Because these benchmarks use small kernels, and because of the reporting options, EEMBC does not have the reputation of being a good predictor of relative performance of different embedded computers in the field. This lack of success is why Dhrystone, which EEMBC was trying to replace, is sadly still used.

One of the most successful attempts to create standardized benchmark application suites has been the SPEC (Standard Performance Evaluation Corporation), which had its roots in efforts in the late 1980s to deliver better benchmarks for workstations. Just as the computer industry has evolved over time, so has the need for different benchmark suites, and there are now SPEC benchmarks to cover many application classes. All the SPEC benchmark suites and their reported results are found at http://www.spec.org.

Although we focus our discussion on the SPEC benchmarks in many of the following sections, many benchmarks have also been developed for PCs running the Windows operating system.

Desktop Benchmarks

Desktop benchmarks divide into two broad classes: processor-intensive benchmarks and graphics-intensive benchmarks, although many graphics benchmarks include intensive processor activity. SPEC originally created a benchmark set focusing on processor performance (initially called SPEC89), which has evolved into its sixth generation: SPEC CPU2017, which follows SPEC2006, SPEC2000, SPEC95 SPEC92, and SPEC89. SPEC CPU2017 consists of a set of 10 integer benchmarks (CINT2017) and 17 floating-point benchmarks (CFP2017). Figure 1.17 describes the current SPEC CPU benchmarks and their ancestry.

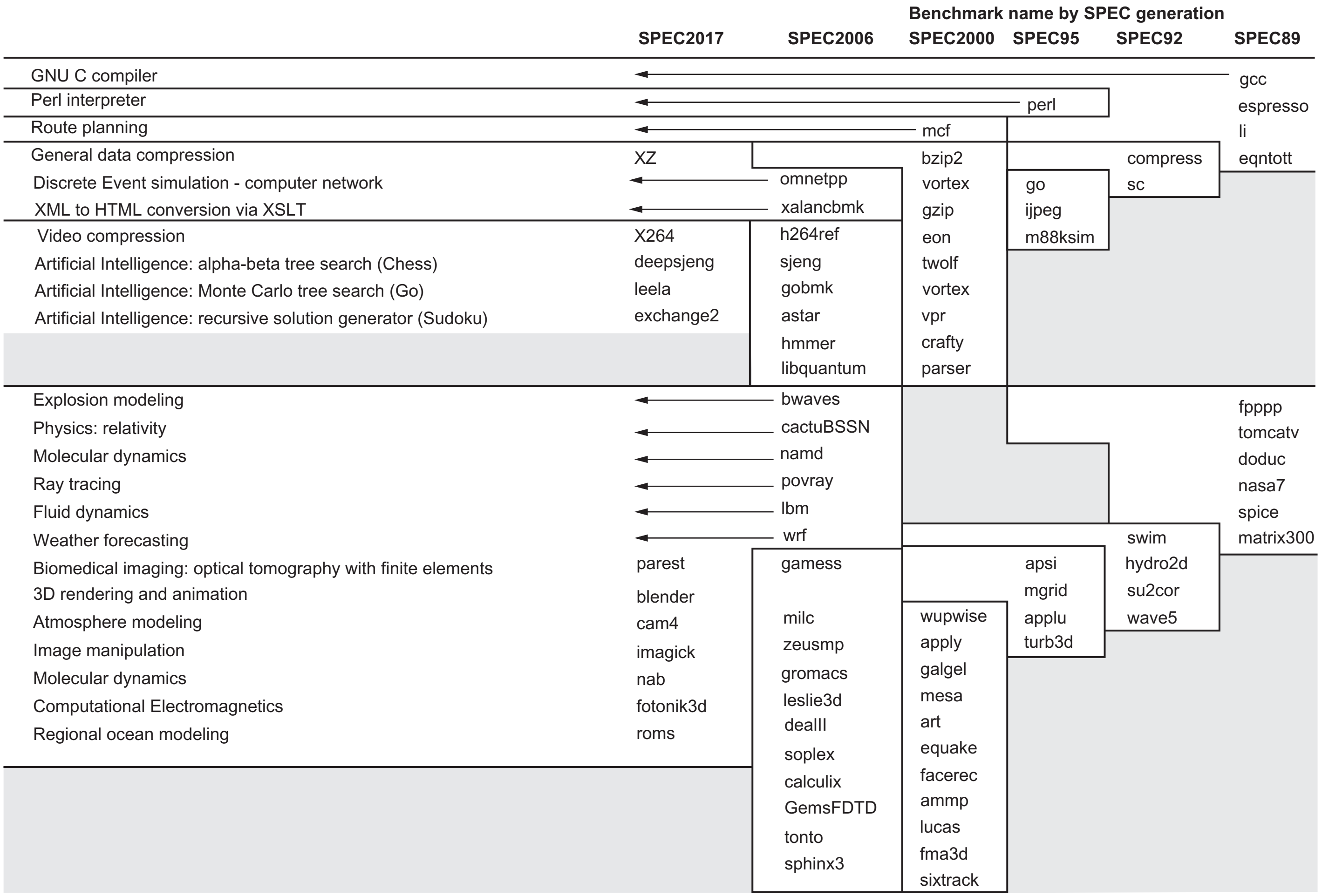

Benchmark name by SPEC generation
SPEC2017
SPEC2006
SPEC2000
SPEC95
SPEC92
SPEC89
GNU C compiler
gcc
Perl interpreter
perl
espresso
Route planning
mcf
li
General data compression
XZ
bzip2
compress
eqntott
Discrete Event simulation - computer network
omnetpp
vortex
go
sc
XML to HTML conversion via XSLT
xalancbmk
gzip
ijpeg
Video compression
X264
h264ref
eon
m88ksim
Artificial Intelligence: alpha-beta tree search (Chess)
deepsjeng
sjeng
twolf
Artificial Intelligence: Monte Carlo tree search (Go)
leela
gobmk
vortex
Artificial Intelligence: recursive solution generator (Sudoku)
exchange2
astar
vpr
hmmer
crafty
libquantum
parser
Explosion modeling
bwaves
fpppp
Physics: relativity
cactuBSSN
tomcatv
Molecular dynamics
namd
doduc
Ray tracing
povray
nasa7
Fluid dynamics
lbm
spice
Weather forecasting
wrf
swim
matrix300
Biomedical imaging: optical tomography with finite elements
parest
gamess
apsi
hydro2d
3D rendering and animation
blender
mgrid
su2cor
Atmosphere modeling
cam4
milc
wupwise
applu
wave5
Image manipulation
imagick
zeusmp
apply
turb3d
Molecular dynamics
nab
gromacs
galgel
Computational Electromagnetics
fotonik3d
leslie3d
mesa
Regional ocean modeling
roms
dealII
art
soplex
equake
calculix
facerec
GemsFDTD
ammp
tonto
lucas
sphinx3
fma3d
sixtrack

Figure 1.17 SPEC2017 programs and the evolution of the SPEC benchmarks over time, with integer programs above the line and floating-point programs below the line. Of the 10 SPEC2017 integer programs, 5 are written in C, 4 in C++., and 1 in Fortran. For the floating-point programs, the split is 3 in Fortran, 2 in C++, 2 in C, and 6 in mixed C, C++, and Fortran. The figure shows all 82 of the programs in the 1989, 1992, 1995, 2000, 2006, and 2017 releases. Gcc is the senior citizen of the group. Only 3 integer programs and 3 floating-point programs survived three or more generations. Although a few are carried over from generation to generation, the version of the program changes and either the input or the size of the benchmark is often expanded to increase its running time and to avoid perturbation in measurement or domination of the execution time by some factor other than CPU time. The benchmark descriptions on the left are for SPEC2017 only and do not apply to earlier versions. Programs in the same row from different generations of SPEC are generally not related; for example, fpppp is not a CFD code like bwaves.

SPEC benchmarks are real programs modified to be portable and to minimize the effect of I/O on performance. The integer benchmarks vary from part of a C compiler to a go program to a video compression. The floating-point benchmarks include molecular dynamics, ray tracing, and weather forecasting. The SPEC CPU suite is useful for processor benchmarking for both desktop systems and single-processor servers. We will see data on many of these programs throughout this book. However, these programs share little with modern programming languages and environments and the Google Translate application that Section 1.1 describes. Nearly half of them are written at least partially in Fortran! They are even statically linked instead of being dynamically linked like most real programs. Alas, the SPEC2017 applications themselves may be real, but they are not inspiring. It's not clear that SPECINT2017 and SPECFP2017 capture what is exciting about computing in the 21st century.

In Section 1.11, we describe pitfalls that have occurred in developing the SPEC CPUbenchmark suite, as well as the challenges in maintaining a useful and predictive benchmark suite.

SPEC CPU2017 is aimed at processor performance, but SPEC offers many other benchmarks. Figure 1.18 lists the 17 SPEC benchmarks that are active in 2017.

Server Benchmarks

Just as servers have multiple functions, so are there multiple types of benchmarks. The simplest benchmark is perhaps a processor throughput-oriented benchmark. SPEC CPU2017 uses the SPEC CPU benchmarks to construct a simple throughput benchmark where the processing rate of a multiprocessor can be measured by running multiple copies (usually as many as there are processors) of each SPEC CPU benchmark and converting the CPU time into a rate. This leads to a measurement called the SPECrate, and it is a measure of request-level parallelism from Section 1.2. To measure thread-level parallelism, SPEC offers what they call high-performance computing benchmarks around OpenMP and MPI as well as for accelerators such as GPUs (see Figure 1.18).

Other than SPECrate, most server applications and benchmarks have significant I/O activity arising from either storage or network traffic, including benchmarks for file server systems, for web servers, and for database and transaction-processing systems. SPEC offers both a file server benchmark (SPECSFS) and a Java server benchmark. (Appendix D discusses some file and I/O system benchmarks in detail.) SPECvirt_Sc2013 evaluates end-to-end performance of virtualized data center servers. Another SPEC benchmark measures power, which we examine in Section 1.10.

Transaction-processing (TP) benchmarks measure the ability of a system to handle transactions that consist of database accesses and updates. Airline reservation systems and bank ATM systems are typical simple examples of TP; more sophisticated TP systems involve complex databases and decision-making.

Category	**Name**	**Measures performance of**
Cloud	Cloud_IaaS 2016	Cloud using NoSQL database transaction and K-Means clustering using map/reduce
CPU	CPU2017	Compute-intensive integer and floating-point workloads
Graphics and workstation performance	SPECviewperf® 12	3D graphics in systems running OpenGL and Direct X
	SPECwpc V2.0	Workstations running professional apps under the Windows OS
	SPECapcSM for 3ds Max 2015™	3D graphics running the proprietary Autodesk 3ds Max 2015 app
	SPECapcSM for Maya® 2012	3D graphics running the proprietary Autodesk 3ds Max 2012 app
	SPECapcSM for PTC Creo 3.0	3D graphics running the proprietary PTC Creo 3.0 app
	SPECapcSM for Siemens NX 9.0 and 10.0	3D graphics running the proprietary Siemens NX 9.0 or 10.0 app
	SPECapcSM for SolidWorks 2015	3D graphics of systems running the proprietary SolidWorks 2015 CAD/CAM app
High performance computing	ACCEL	Accelerator and host CPU running parallel applications using OpenCL and OpenACC
	MPI2007	MPI-parallel, floating-point, compute-intensive programs running on clusters and SMPs
	OMP2012	Parallel apps running OpenMP
Java client/server	SPECjbb2015	Java servers
Power	SPECpower_ssj2008	Power of volume server class computers running SPECjbb2015
Solution File Server (SFS)	SFS2014	File server throughput and response time
	SPECsfs2008	File servers utilizing the NFSv3 and CIFS protocols
Virtualization	SPECvirt_sc2013	Datacenter servers used in virtualized server consolidation

Figure 1.18 Active benchmarks from SPEC as of 2017.

In the mid-1980s, a group of concerned engineers formed the vendor-independent Transaction Processing Council (TPC) to try to create realistic and fair benchmarks for TP. The TPC benchmarks are described at http://www.tpc.org.

The first TPC benchmark, TPC-A, was published in 1985 and has since been replaced and enhanced by several different benchmarks. TPC-C, initially created in 1992, simulates a complex query environment. TPC-H models ad hoc decision support—the queries are unrelated and knowledge of past queries cannot be used to optimize future queries. The TPC-DI benchmark, a new data integration (DI) task also known as ETL, is an important part of data warehousing. TPC-E is an online transaction processing (OLTP) workload that simulates a brokerage firm's customer accounts.

Recognizing the controversy between traditional relational databases and “No SQL” storage solutions, TPCx-HS measures systems using the Hadoop file system running MapReduce programs, and TPC-DS measures a decision support system that uses either a relational database or a Hadoop-based system. TPC-VMS and TPCx-V measure database performance for virtualized systems, and TPC-Energy adds energy metrics to all the existing TPC benchmarks.

All the TPC benchmarks measure performance in transactions per second. In addition, they include a response time requirement so that throughput performance is measured only when the response time limit is met. To model real-world systems, higher transaction rates are also associated with larger systems, in terms of both users and the database to which the transactions are applied. Finally, the system cost for a benchmark system must be included as well to allow accurate comparisons of cost-performance. TPC modified its pricing policy so that there is a single specification for all the TPC benchmarks and to allow verification of the prices that TPC publishes.

Reporting Performance Results

The guiding principle of reporting performance measurements should be *reproducibility*—list everything another experimenter would need to duplicate the results. A SPEC benchmark report requires an extensive description of the computer and the compiler flags, as well as the publication of both the baseline and the optimized results. In addition to hardware, software, and baseline tuning parameter descriptions, a SPEC report contains the actual performance times, shown both in tabular form and as a graph. A TPC benchmark report is even more complete, because it must include results of a benchmarking audit and cost information. These reports are excellent sources for finding the real costs of computing systems, since manufacturers compete on high performance and cost-performance.

Summarizing Performance Results

In practical computer design, one must evaluate myriad design choices for their relative quantitative benefits across a suite of benchmarks believed to be relevant. Likewise, consumers trying to choose a computer will rely on performance measurements from benchmarks, which ideally are similar to the users’ applications. In both cases, it is useful to have measurements for a suite of benchmarks so that the performance of important applications is similar to that of one or more benchmarks in the suite and so that variability in performance can be understood. In the best case, the suite resembles a statistically valid sample of the application space, but such a sample requires more benchmarks than are typically found in most suites and requires a randomized sampling, which essentially no benchmark suite uses.

Once we have chosen to measure performance with a benchmark suite, we want to be able to summarize the performance results of the suite in a unique number. A simple approach to computing a summary result would be to compare the arithmetic means of the execution times of the programs in the suite. An alternative would be to add a weighting factor to each benchmark and use the weighted arithmetic mean as the single number to summarize performance. One approach is to use weights that make all programs execute an equal time on some reference computer, but this biases the results toward the performance characteristics of the reference computer.

Rather than pick weights, we could normalize execution times to a reference computer by dividing the time on the reference computer by the time on the computer being rated, yielding a ratio proportional to performance. SPEC uses this approach, calling the ratio the SPECRatio. It has a particularly useful property that matches the way we benchmark computer performance throughout this text—namely, comparing performance ratios. For example, suppose that the SPECRatio of computer A on a benchmark is 1.25 times as fast as computer B; then we know

$$1.25 = \frac{\text{SPECRatio}_{\text{A}}}{\text{SPECRatio}_{\text{B}}} = \frac{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_{\text{A}}}}{\dfrac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_{\text{B}}}} = \frac{\text{Execution time}_{\text{B}}}{\text{Execution time}_{\text{A}}} = \frac{\text{Performance}_{\text{A}}}{\text{Performance}_{\text{B}}}$$

Notice that the execution times on the reference computer drop out and the choice of the reference computer is irrelevant when the comparisons are made as a ratio, which is the approach we consistently use. Figure 1.19 gives an example.

Because a SPECRatio is a ratio rather than an absolute execution time, the mean must be computed using the *geometric* mean. (Because SPECRatios have no units, comparing SPECRatios arithmetically is meaningless.) The formula is

$$\text{Geometric mean} = \sqrt[n]{\prod_{i=1}^{n} sample_i}$$

In the case of SPEC, $sample_i$ is the SPECRatio for program i. Using the geometric mean ensures two important properties:

1. The geometric mean of the ratios is the same as the ratio of the geometric means.
2. The ratio of the geometric means is equal to the geometric mean of the performance ratios, which implies that the choice of the reference computer is irrelevant.

Therefore the motivations to use the geometric mean are substantial, especially when we use performance ratios to make comparisons.

Example Show that the ratio of the geometric means is equal to the geometric mean of the performance ratios and that the reference computer of SPECRatio does not matter.

Answer Assume two computers A and B and a set of SPECRatios for each.

$$\frac{\text{Geometric mean}_{\text{A}}}{\text{Geometric mean}_{\text{B}}} = \frac{\sqrt[n]{\prod_{i=1}^{n}\text{SPECRatio A}_i}}{\sqrt[n]{\prod_{i=1}^{n}\text{SPECRatio B}_i}} = \sqrt[n]{\prod_{i=1}^{n}\frac{\text{SPECRatio A}_i}{\text{SPECRatio B}_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n}\frac{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{\text{A}_i}}}{\dfrac{\text{Execution time}_{\text{reference}_i}}{\text{Execution time}_{\text{B}_i}}}} = \sqrt[n]{\prod_{i=1}^{n}\frac{\text{Execution time}_{\text{B}_i}}{\text{Execution time}_{\text{A}_i}}} = \sqrt[n]{\prod_{i=1}^{n}\frac{\text{Performance}_{\text{A}_i}}{\text{Performance}_{\text{B}_i}}}$$

That is, the ratio of the geometric means of the SPECRatios of A and B is the geometric mean of the performance ratios of A to B of all the benchmarks in the suite. Figure 1.19 demonstrates this validity using examples from SPEC.

Benchmarks	Sun Ultra Enterprise 2 time (seconds)	AMD A10-6800K time (seconds)	SPEC 2006Cint ratio	Intel Xeon E5-2690 time (seconds)	SPEC 2006Cint ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	9770	401	24.36	261	37.43	1.54	1.54
bzip2	9650	505	19.11	422	22.87	1.20	1.20
gcc	8050	490	16.43	227	35.46	2.16	2.16
mcf	9120	249	36.63	153	59.61	1.63	1.63
gobmk	10,490	418	25.10	382	27.46	1.09	1.09
hmmer	9330	182	51.26	120	77.75	1.52	1.52
sjeng	12,100	517	23.40	383	31.59	1.35	1.35
libquantum	20,720	84	246.08	3	7295.77	29.65	29.65
h264ref	22,130	611	36.22	425	52.07	1.44	1.44
omnetpp	6250	313	19.97	153	40.85	2.05	2.05
astar	7020	303	23.17	209	33.59	1.45	1.45
xalancbmk	6900	215	32.09	98	70.41	2.19	2.19
Geometric mean			31.91		63.72	2.00	2.00

Figure 1.19 SPEC2006Cint execution times (in seconds) for the Sun Ultra 5—the reference computer of SPEC2006—and execution times and SPECRatios for the AMD A10 and Intel Xeon E5-2690. The final two columns show the ratios of execution times and SPEC ratios. This figure demonstrates the irrelevance of the reference computer in relative performance. The ratio of the execution times is identical to the ratio of the SPEC ratios, and the ratio of the geometric means (63.7231.91/20.86 = 2.00) is identical to the geometric mean of the ratios (2.00). Section 1.11 discusses libquantum, whose performance is orders of magnitude higher than the other SPEC benchmarks.

1.9 Quantitative Principles of Computer Design

Now that we have seen how to define, measure, and summarize performance, cost, dependability, energy, and power, we can explore guidelines and principles that are useful in the design and analysis of computers. This section introduces important observations about design, as well as two equations to evaluate alternatives.

Take Advantage of Parallelism

Using parallelism is one of the most important methods for improving performance. Every chapter in this book has an example of how performance is enhanced through the exploitation of parallelism. We give three brief examples here, which are expounded on in later chapters.

Our first example is the use of parallelism at the system level. To improve the throughput performance on a typical server benchmark, such as SPECSFS or TPC-C, multiple processors and multiple storage devices can be used. The workload of handling requests can then be spread among the processors and storage devices, resulting in improved throughput. Being able to expand memory and the number of processors and storage devices is called *scalability*, and it is a valuable asset for servers. Spreading of data across many storage devices for parallel reads and writes enables data-level parallelism. SPECSFS also relies on request-level parallelism to use many processors, whereas TPC-C uses thread-level parallelism for faster processing of database queries.

At the level of an individual processor, taking advantage of parallelism among instructions is critical to achieving high performance. One of the simplest ways to do this is through pipelining. (Pipelining is explained in more detail in Appendix C and is a major focus of Chapter 3.) The basic idea behind pipelining is to overlap instruction execution to reduce the total time to complete an instruction sequence. A key insight into pipelining is that not every instruction depends on its immediate predecessor, so executing the instructions completely or partially in parallel may be possible. Pipelining is the best-known example of ILP.

Parallelism can also be exploited at the level of detailed digital design. For example, set-associative caches use multiple banks of memory that are typically searched in parallel to find a desired item. Arithmetic-logical units use carry-lookahead, which uses parallelism to speed the process of computing sums from linear to logarithmic in the number of bits per operand. These are more examples of *data-level parallelism.*

Principle of Locality

Important fundamental observations have come from properties of programs. The most important program property that we regularly exploit is the *principle of locality*: programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses.

Two different types of locality have been observed. *Temporal locality* states that recently accessed items are likely to be accessed soon. *Spatial locality* says that items whose addresses are near one another tend to be referenced close together in time. We will see these principles applied in Chapter 2.

Focus on the Common Case

Perhaps the most important and pervasive principle of computer design is to focus on the common case: in making a design trade-off, favor the frequent case over the infrequent case. This principle applies when determining how to spend resources, because the impact of the improvement is higher if the occurrence is commonplace.

Focusing on the common case works for energy as well as for resource allocation and performance. The instruction fetch and decode unit of a processor may be used much more frequently than a multiplier, so optimize it first. It works on dependability as well. If a database server has 50 storage devices for every processor, storage dependability will dominate system dependability.

In addition, the common case is often simpler and can be done faster than the infrequent case. For example, when adding two numbers in the processor, we can expect overflow to be a rare circumstance and can therefore improve performance by optimizing the more common case of no overflow. This emphasis may slow down the case when overflow occurs, but if that is rare, then overall performance will be improved by optimizing for the normal case.

We will see many cases of this principle throughout this text. In applying this simple principle, we have to decide what the frequent case is and how much performance can be improved by making that case faster. A fundamental law, called *Amdahl's Law*, can be used to quantify this principle.

Amdahl's Law

The performance gain that can be obtained by improving some portion of a computer can be calculated using Amdahl's Law. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

Amdahl's Law defines the *speedup* that can be gained by using a particular feature. What is speedup? Suppose that we can make an enhancement to a computer that will improve performance when it is used. Speedup is the ratio

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

Speedup tells us how much faster a task will run using the computer with the enhancement contrary to the original computer.

Amdahl's Law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement*—For example, if 40 seconds of the execution time of a program that takes 100 seconds in total can use an enhancement, the fraction is 40/100. This value, which we call $\text{Fraction}_{\text{enhanced}}$, is always less than or equal to 1.
2. *The improvement gained by the enhanced execution mode, that is, how much faster the task would run if the enhanced mode were used for the entire program*—This value is the time of the original mode over the time of the enhanced mode. If the enhanced mode takes, say, 4 seconds for a portion of the program, while it is 40 seconds in the original mode, the improvement is 40/4 or 10. We call this value, which is always greater than 1, $\text{Speedup}_{\text{enhanced}}$.

The execution time using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Example Suppose that we want to enhance the processor used for web serving. The new processor is 10 times faster on computation in the web serving application than the old processor. Assuming that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer

$$\text{Fraction}_{\text{enhanced}} = 0.4;\ \text{Speedup}_{\text{enhanced}} = 10;\ \text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \dfrac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

Amdahl's Law expresses the law of diminishing returns: The incremental improvement in speedup gained by an improvement of just a portion of the computation diminishes as improvements are added. An important corollary of Amdahl's Law is that if an enhancement is usable only for a fraction of a task, then we can't speed up the task by more than the reciprocal of 1 minus that fraction.

A common mistake in applying Amdahl's Law is to confuse "fraction of time converted *to use an enhancement*" and "fraction of time *after enhancement is in use*." If, instead of measuring the time that we *could use* the enhancement in a computation, we measure the time *after* the enhancement is in use, the results will be incorrect!

Amdahl's Law can serve as a guide to how much an enhancement will improve performance and how to distribute resources to improve cost-performance. The goal, clearly, is to spend resources proportional to where time is spent. Amdahl's Law is particularly useful for comparing the overall system performance of two alternatives, but it can also be applied to compare two processor design alternatives, as the following example shows.

Example A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FSQRT) is responsible for 20% of the execution time of a critical graphics benchmark. One proposal is to enhance the FSQRT hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FSQRT}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

Amdahl's Law is applicable beyond performance. Let's redo the reliability example from page 39 after improving the reliability of the power supply via redundancy from 200,000-hour to 830,000,000-hour MTTF, or 4150 × better.

Example The calculation of the failure rates of the disk subsystem was

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10+2+5+5+1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000 \text{ hours}}$$

Therefore the fraction of the failure rate that could be improved is 5 per million hours out of 23 for the whole system, or 0.22.

Answer The reliability improvement would be

$$\text{Improvement}_{\text{power supply pair}} = \frac{1}{(1-0.22)+\dfrac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

Despite an impressive 4150 × improvement in reliability of one module, from the system's perspective, the change has a measurable but small benefit.

In the preceding examples, we needed the fraction consumed by the new and improved version; often it is difficult to measure these times directly. In the next section, we will see another way of doing such comparisons based on the use of an equation that decomposes the CPU execution time into three separate components. If we know how an alternative affects these three components, we can determine its overall performance. Furthermore, it is often possible to build simulators that measure these components before the hardware is actually designed.

The Processor Performance Equation

Essentially all computers are constructed using a clock running at a constant rate. These discrete time events are called *clock periods*, *clocks*, *cycles*, or *clock cycles*. Computer designers refer to the time of a clock period by its duration (e.g., 1 ns) or by its rate (e.g., 1 GHz). CPU time for a program can then be expressed two ways:

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

or

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

In addition to the number of clock cycles needed to execute a program, we can also count the number of instructions executed—the *instruction path length* or *instruction count* (IC). If we know the number of clock cycles and the instruction count, we can calculate the average number of *clock cycles per instruction* (CPI). Because it is easier to work with, and because we will deal with simple processors in this chapter, we use CPI. Designers sometimes also use *instructions per clock* (IPC), which is the inverse of CPI.

CPI is computed as

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

This processor figure of merit provides insight into different styles of instruction sets and implementations, and we will use it extensively in the next four chapters.

By transposing the instruction count in the preceding formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the execution time formula:

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Clock cycle time}$$

Expanding the first formula into the units of measurement shows how the pieces fit together:

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

As this formula demonstrates, processor performance is dependent upon three characteristics: clock cycle (or rate), clock cycles per instruction, and instruction count. Furthermore, CPU time is *equally* dependent on these three characteristics; for example, a 10% improvement in any one of them leads to a 10% improvement in CPU time.

Unfortunately, it is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

Luckily, many potential performance improvement techniques primarily enhance one component of processor performance with small or predictable impacts on the other two.

In designing the processor, sometimes it is useful to calculate the number of total processor clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

where IC_i represents the number of times instruction i is executed in a program and CPI_i represents the average number of clocks per instruction for instruction i. This form can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i\right) \times \text{Clock cycle time}$$

and overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $\text{IC}_i \div \text{Instruction count}$). Because it must include pipeline effects, cache misses, and any other memory system inefficiencies, CPI_i should be measured and not just calculated from a table in the back of a reference manual.

Consider our performance example on page 52, here modified to use measurements of the frequency of the instructions and of the instruction CPI values, which, in practice, are obtained by simulation or by hardware instrumentation.

Example Suppose we made the following measurements:

Frequency of FP operations = 25%
Average CPI of FP operations = 4.0
Average CPI of other instructions = 1.33
Frequency of FSQRT = 2%
CPI of FSQRT = 20

Assume that the two design alternatives are to decrease the CPI of FSQRT to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the processor performance equation.

Answer First, observe that only the CPI changes; the clock rate and instruction count remain identical. We start by finding the original CPI with neither enhancement:

$$\begin{aligned}\text{CPI}_{\text{original}} &= \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}}\right)\\ &= (4 \times 25\%) + (1.33 \times 75\%) = 2.0\end{aligned}$$

We can compute the CPI for the enhanced FSQRT by subtracting the cycles saved from the original CPI:

$$\begin{aligned}\text{CPI}_{\text{with new FPSQR}} &= \text{CPI}_{\text{original}} - 2\% \times \left(\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}}\right)\\ &= 2.0 - 2\% \times (20 - 2) = 1.64\end{aligned}$$

We can compute the CPI for the enhancement of all FP instructions the same way or by summing the FP and non-FP CPIs. Using the latter gives us

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better. Specifically, the speedup for the overall FP enhancement is

$$\begin{aligned}\text{Speedup}_{\text{new FP}} &= \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}\\ &= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23\end{aligned}$$

Happily, we obtained this same speedup using Amdahl's Law on page 51.

It is often possible to measure the constituent parts of the processor performance equation. Such isolated measurements are a key advantage of using the processor performance equation versus Amdahl's Law in the previous example. In particular, it may be difficult to measure things such as the fraction of execution time for which a set of instructions is responsible. In practice, this would probably be computed by summing the product of the instruction count and the CPI for each of the instructions in the set. Since the starting point is often individual instruction count and CPI measurements, the processor performance equation is incredibly useful.

To use the processor performance equation as a design tool, we need to be able to measure the various factors. For an existing processor, it is easy to obtain the execution time by measurement, and we know the default clock speed. The challenge lies in discovering the instruction count or the CPI. Most processors include counters for both instructions executed and clock cycles. By periodically monitoring these counters, it is also possible to attach execution time and instruction count to segments of the code, which can be helpful to programmers trying to understand and tune the performance of an application. Often designers or programmers will want to understand performance at a more fine-grained level than what is available from the hardware counters. For example, they may want to know why the CPI is what it is. In such cases, the simulation techniques used are like those for processors that are being designed.

Techniques that help with energy efficiency, such as dynamic voltage frequency scaling and overclocking (see Section 1.5), make this equation harder to use, because the clock speed may vary while we measure the program. A simple approach is to turn off those features to make the results reproducible. Fortunately, as performance and energy efficiency are often highly correlated—taking less time to run a program generally saves energy—it's probably safe to consider performance without worrying about the impact of DVFS or overclocking on the results.

1.10 Putting It All Together: Performance, Price, and Power

In the "Putting It All Together" sections that appear near the end of every chapter, we provide real examples that use the principles in that chapter. In this section, we look at measures of performance and power-performance in small servers using the SPECpower benchmark.

Figure 1.20 shows the three multiprocessor servers we are evaluating along with their price. To keep the price comparison fair, all are Dell PowerEdge servers. The first is the PowerEdge R710, which is based on the Intel Xeon ×85670 microprocessor with a clock rate of 2.93 GHz. Unlike the Intel Core i7-6700 in Chapters 2–5, which has 20 cores and a 40 MB L3 cache, this Intel chip has 22 cores and a 55 MB L3 cache, although the cores themselves are identical. We selected a two-socket system—so 44 cores total—with 128 GB of ECC-protected 2400 MHz DDR4 DRAM. The next server is the PowerEdge C630, with the same processor, number of sockets, and DRAM. The main difference is a smaller rack-mountable package: "2U" high (3.5 inches) for the 730 versus "1U" (1.75 inches) for the 630.

	System 1		System 2		System 3	
Component		**Cost (% Cost)**		**Cost (% Cost)**		**Cost (% Cost)**
Base server	PowerEdge R710	$653 (7%)	PowerEdge R815	$1437 (15%)	PowerEdge R815	$1437 (11%)
Power supply	570 W		1100 W		1100 W	
Processor	Xeon X5670	$3738 (40%)	Opteron 6174	$2679 (29%)	Opteron 6174	$5358 (42%)
Clock rate	2.93 GHz		2.20 GHz		2.20 GHz	
Total cores	12		24		48	
Sockets	2		2		4	
Cores/socket	6		12		12	
DRAM	12 GB	$484 (5%)	16 GB	$693 (7%)	32 GB	$1386 (11%)
Ethernet Inter.	Dual 1-Gbit	$199 (2%)	Dual 1-Gbit	$199 (2%)	Dual 1-Gbit	$199 (2%)
Disk	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (14%)	50 GB SSD	$1279 (10%)
Windows OS		$2999 (32%)		$2999 (33%)		$2999 (24%)
Total		$9352 (100%)		$9286 (100%)		$12,658 (100%)
Max ssj_ops	910,978		926,676		1,840,450	
Max ssj_ops/$	97		100		145	

Figure 1.20 Three Dell PowerEdge servers being measured and their prices as of July 2016. We calculated the cost of the processors by subtracting the cost of a second processor. Similarly, we calculated the overall cost of memory by seeing what the cost of extra memory was. Hence the base cost of the server is adjusted by removing the estimated cost of the default processor and memory. Chapter 5 describes how these multisocket systems are connected together, and Chapter 6 describes how clusters are connected together.

The third server is a cluster of 16 of the PowerEdge 630 s that is connected together with a 1 Gbit/s Ethernet switch. All are running the Oracle Java HotSpot version 1.7 Java Virtual Machine (JVM) and the Microsoft Windows Server 2012 R2 Datacenter version 6.3 operating system.

Note that because of the forces of benchmarking (see Section 1.11), these are unusually configured servers. The systems in Figure 1.20 have little memory relative to the amount of computation, and just a tiny 120 GB solid-state disk. It is inexpensive to add cores if you don't need to add commensurate increases in memory and storage!

Rather than run statically linked C programs of SPEC CPU, SPECpower uses a more modern software stack written in Java. It is based on SPECjbb, and it represents the server side of business applications, with performance measured as the number of transactions per second, called *ssj_ops* for *server side Java operations per second*. It exercises not only the processor of the server, as does SPEC CPU, but also the caches, memory system, and even the multiprocessor interconnection system. In addition, it exercises the JVM, including the JIT runtime compiler and garbage collector, as well as portions of the underlying operating system.

As the last two rows of Figure 1.20 show, the performance winner is the cluster of 16 R630s, which is hardly a surprise since it is by far the most expensive. The price-performance winner is the PowerEdge R630, but it barely beats the cluster at 213 versus 211 ssj-ops/$. Amazingly, the 16 node cluster is within 1% of the same price-performances of a single node despite being 16 times as large.

While most benchmarks (and most computer architects) care only about performance of systems at peak load, computers rarely run at peak load. Indeed, Figure 6.2 in Chapter 6 shows the results of measuring the utilization of tens of thousands of servers over 6 months at Google, and less than 1% operate at an average utilization of 100%. The majority have an average utilization of between 10% and 50%. Thus the SPECpower benchmark captures power as the target workload varies from its peak in 10% intervals all the way to 0%, which is called Active Idle.

Figure 1.21 plots the ssj_ops (SSJ operations/second) per watt and the average power as the target load varies from 100% to 0%. The Intel R730 always has the lowest power and the single node R630 has the best ssj_ops per watt across each target workload level. Since watts = joules/second, this metric is proportional to SSJ operations per joule:

$$\frac{\text{ssj\_operations/second}}{\text{Watt}} = \frac{\text{ssj\_operations/second}}{\text{Joule/second}} = \frac{\text{ssj\_operations}}{\text{Joule}}$$

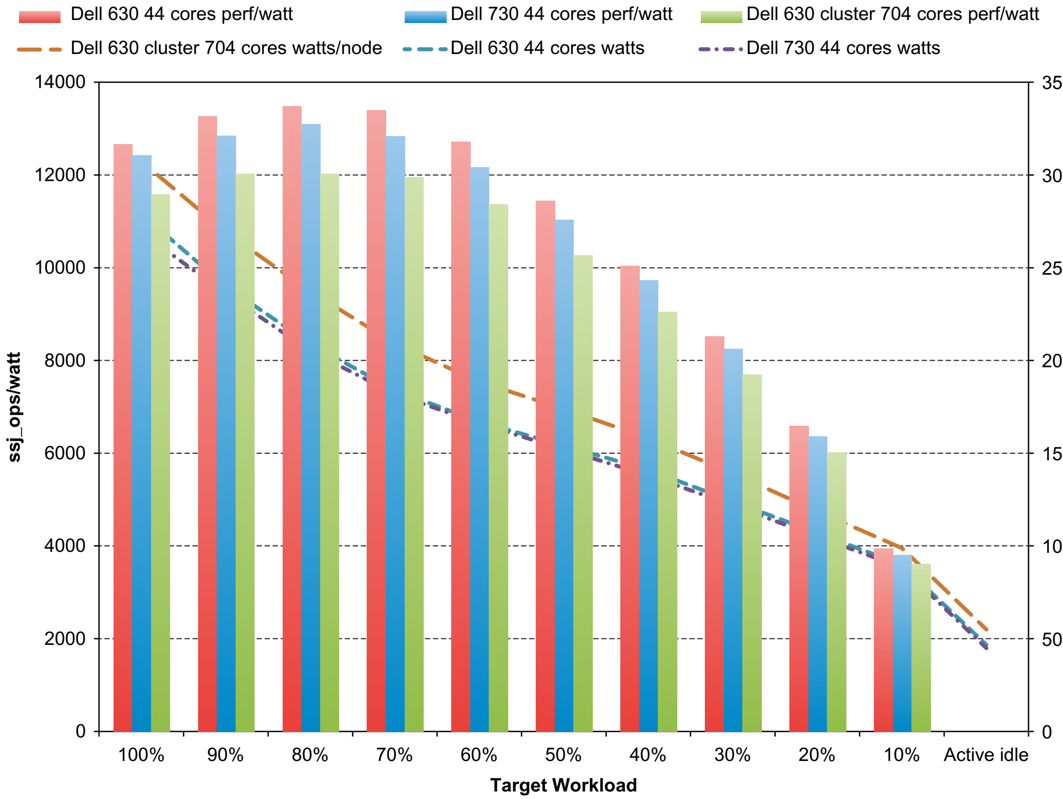


Figure 1.21 Power-performance of the three servers in Figure 1.20. Ssj_ops/watt values are on the left axis, with the three columns associated with it, and watts are on the right axis, with the three lines associated with it. The horizontal axis shows the target workload, as it varies from 100% to Active Idle. The single node R630 has the best ssj_ops/watt at each workload level, but R730 consumes the lowest power at each level.

To calculate a single number to use to compare the power efficiency of systems, SPECpower uses

$$\text{Overall ssj\_ops/watt} = \frac{\sum \text{ssj\_ops}}{\sum \text{power}}$$

The overall ssj_ops/watt of the three servers is 10,802 for the R730, 11,157 for the R630, and 10,062 for the cluster of 16 R630s. Therefore the single node R630 has the best power-performance. Dividing by the price of the servers, the ssj_ops/watt/\$1,000 is 879 for the R730, 899 for the R630, and 789 (per node) for the 16-node cluster of R630s. Thus, after adding power, the single-node R630 is still in first place in performance/price, but now the single-node R730 is significantly more efficient than the 16-node cluster.

1.11 Fallacies and Pitfalls

The purpose of this section, which will be found in every chapter, is to explain some commonly held misbeliefs or misconceptions that you should avoid. We call such misbeliefs *fallacies.* When discussing a fallacy, we try to give a counterexample. We also discuss *pitfalls*—easily made mistakes. Often pitfalls are generalizations of principles that are true in a limited context. The purpose of these sections is to help you avoid making these errors in computers that you design.

Pitfall *All exponential laws must come to an end.*

The first to go was Dennard scaling. Dennard's 1974 observation was that power density was constant as transistors got smaller. If a transistor's linear region shrank by a factor 2, then both the current and voltage were also reduced by a factor of 2, and so the power it used fell by 4. Thus chips could be designed to operate faster and still use less power. Dennard scaling ended 30 years after it was observed, not because transistors didn't continue to get smaller but because integrated circuit dependability limited how far current and voltage could drop. The threshold voltage was driven so low that static power became a significant fraction of overall power.

The next deceleration was hard disk drives. Although there was no law for disks, in the past 30 years the maximum areal density of hard drives—which determines disk capacity—improved by 30%–100% per year. In more recent years, it has been less than 5% per year. Increasing density per drive has come primarily from adding more platters to a hard disk drive.

Next up was the venerable Moore's Law. It's been a while since the number of transistors per chip doubled every one to two years. For example, the DRAM chip introduced in 2014 contained 8B transistors, and we won't have a 16B transistor DRAM chip in mass production until 2019, but Moore's Law predicts a 64B transistor DRAM chip.

Moreover, the actual end of scaling of the planar logic transistor was even predicted to end by 2021. Figure 1.22 shows the predictions of the physical gate length

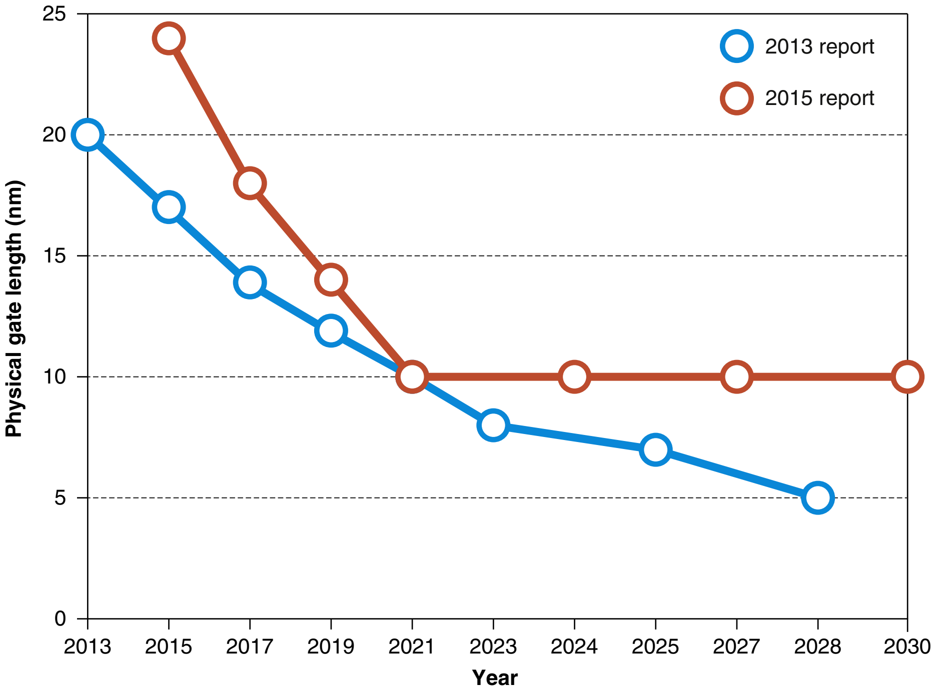


Figure 1.22 Predictions of logic transistor dimensions from two editions of the ITRS report. These reports started in 2001, but 2015 will be the last edition, as the group has disbanded because of waning interest. The only companies that can produce state-of-the-art logic chips today are GlobalFoundaries, Intel, Samsung, and TSMC, whereas there were 19 when the first ITRS report was released. With only four companies left, sharing of plans was too hard to sustain. From IEEE Spectrum, July 2016, "Transistors will stop shrinking in 2021, Moore's Law Roadmap Predicts," by Rachel Courtland.

of the logic transistor from two editions of the International Technology Roadmap for Semiconductors (ITRS). Unlike the 2013 report that projected gate lengths to reach 5 nm by 2028, the 2015 report projects the length stopping at 10 nm by 2021. Density improvements thereafter would have to come from ways other than shrinking the dimensions of transistors. It's not as dire as the ITRS suggests, as companies like Intel and TSMC have plans to shrink to 3 nm gate lengths, but the rate of change is decreasing.

Figure 1.23 shows the changes in increases in bandwidth over time for microprocessors and DRAM—which are affected by the end of Dennard scaling and Moore's Law—as well as for disks. The slowing of technology improvements is apparent in the dropping curves. The continued networking improvement is due to advances in fiber optics and a planned change in pulse amplitude modulation (PAM-4) allowing two-bit encoding so as to transmit information at 400 Gbit/s.

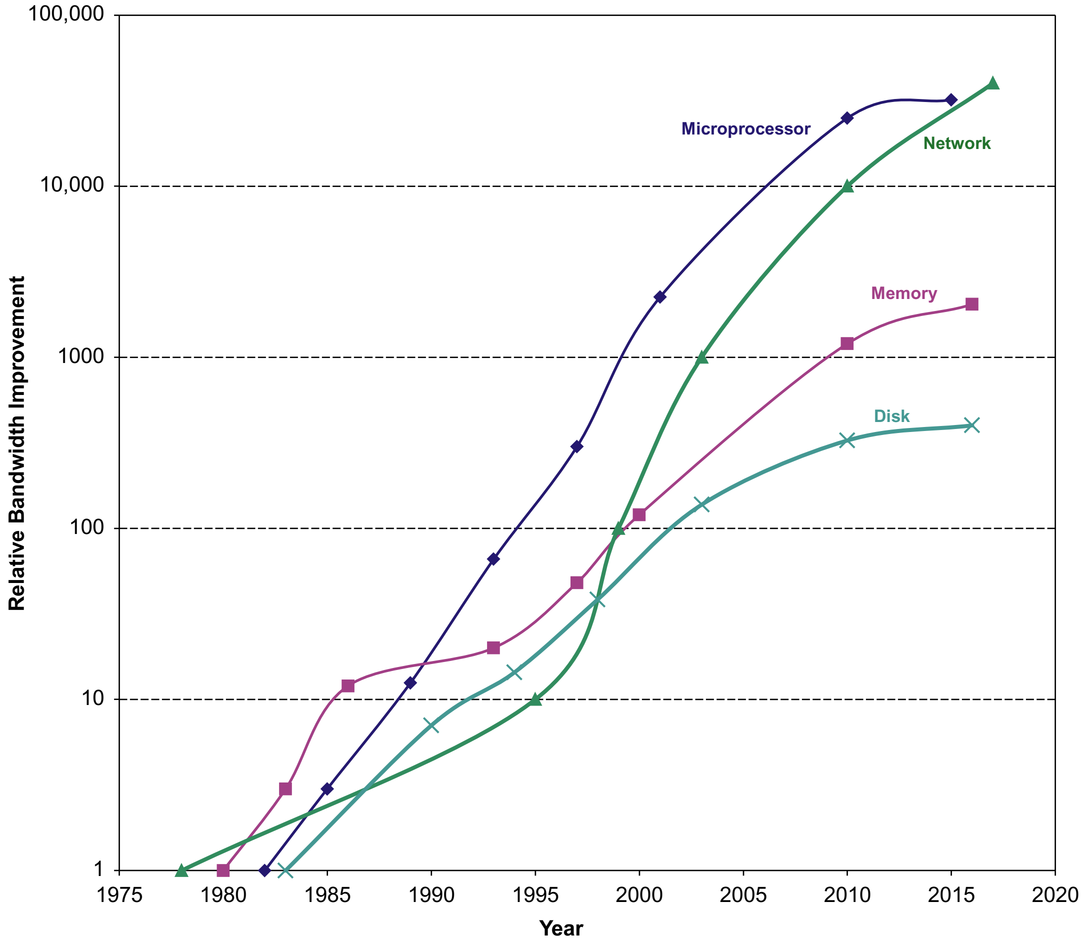


Figure 1.23 Relative bandwidth for microprocessors, networks, memory, and disks over time, based on data in Figure 1.10.

Fallacy *Multiprocessors are a silver bullet.*

The switch to multiple processors per chip around 2005 did not come from some breakthrough that dramatically simplified parallel programming or made it easy to build multicore computers. The change occurred because there was no other option due to the ILP walls and power walls. Multiple processors per chip do not guarantee lower power; it's certainly feasible to design a multicore chip that uses more power. The potential is just that it's possible to continue to improve performance by replacing a high-clock-rate, inefficient core with several lower-clock-rate, efficient cores. As technology to shrink transistors improves, it can shrink both capacitance and the supply voltage a bit so that we can get a modest increase in the number of cores per generation. For example, for the past few years, Intel has been adding two cores per generation in their higher-end chips.

As we will see in Chapters 4 and 5, performance is now a programmer's burden. The programmers' La-Z-Boy era of relying on a hardware designer to make their programs go faster without lifting a finger is officially over. If programmers want their programs to go faster with each generation, they must make their programs more parallel.

The popular version of Moore's law—increasing performance with each generation of technology—is now up to programmers.

Pitfall *Falling prey to Amdahl's heartbreaking law.*

Virtually every practicing computer architect knows Amdahl's Law. Despite this, we almost all occasionally expend tremendous effort optimizing some feature before we measure its usage. Only when the overall speedup is disappointing do we recall that we should have measured first before we spent so much effort enhancing it!

Pitfall *A single point of failure.*

The calculations of reliability improvement using Amdahl's Law on page 53 show that dependability is no stronger than the weakest link in a chain. No matter how much more dependable we make the power supplies, as we did in our example, the single fan will limit the reliability of the disk subsystem. This Amdahl's Law observation led to a rule of thumb for fault-tolerant systems to make sure that every component was redundant so that no single component failure could bring down the whole system. Chapter 6 shows how a software layer avoids single points of failure inside WSCs.

Fallacy *Hardware enhancements that increase performance also improve energy efficiency, or are at worst energy neutral.*

Esmaeilzadeh et al. (2011) measured SPEC2006 on just one core of a 2.67 GHz Intel Core i7 using Turbo mode (Section 1.5). Performance increased by a factor of 1.07 when the clock rate increased to 2.94 GHz (or a factor of 1.10), but the i7 used a factor of 1.37 more joules and a factor of 1.47 more watt hours!

Fallacy *Benchmarks remain valid indefinitely.*

Several factors influence the usefulness of a benchmark as a predictor of real performance, and some change over time. A big factor influencing the usefulness of a benchmark is its ability to resist "benchmark engineering" or "benchmarketing." Once a benchmark becomes standardized and popular, there is tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark. Short kernels or programs that spend their time in a small amount of code are particularly vulnerable.

For example, despite the best intentions, the initial SPEC89 benchmark suite included a small kernel, called matrix300, which consisted of eight different 300 × 300 matrix multiplications. In this kernel, 99% of the execution time was in a single line (see SPEC, 1989). When an IBM compiler optimized this inner loop (using a good idea called *blocking*, discussed in Chapters 2 and 4), performance improved by a factor of 9 over a prior version of the compiler! This benchmark tested compiler tuning and was not, of course, a good indication of overall performance, nor of the typical value of this particular optimization.

Figure 1.19 shows that if we ignore history, we may be forced to repeat it. SPEC Cint2006 had not been updated for a decade, giving compiler writers substantial time to hone their optimizers to this suite. Note that the SPEC ratios of all benchmarks but libquantum fall within the range of 16–52 for the AMD computer and from 22 to 78 for Intel. Libquantum runs about 250 times faster on AMD and 7300 times faster on Intel! This "miracle" is a result of optimizations by the Intel compiler that automatically parallelizes the code across 22 cores and optimizes memory by using bit packing, which packs together multiple narrow-range integers to save memory space and thus memory bandwidth. If we drop this benchmark and recalculate the geometric means, AMD SPEC Cint2006 falls from 31.9 to 26.5 and Intel from 63.7 to 41.4. The Intel computer is now about 1.5 times as fast as the AMD computer instead of 2.0 if we include libquantum, which is surely closer to their real relative performances. SPECCPU2017 dropped libquantum.

To illustrate the short lives of benchmarks, Figure 1.17 on page 43 lists the status of all 82 benchmarks from the various SPEC releases; Gcc is the lone survivor from SPEC89. Amazingly, about 70% of all programs from SPEC2000 or earlier were dropped from the next release.

Fallacy *The rated mean time to failure of disks is 1,200,000 hours or almost 140 years, so disks practically never fail.*

The current marketing practices of disk manufacturers can mislead users. How is such an MTTF calculated? Early in the process, manufacturers will put thousands of disks in a room, run them for a few months, and count the number that fail. They compute MTTF as the total number of hours that the disks worked cumulatively divided by the number that failed.

One problem is that this number far exceeds the lifetime of a disk, which is commonly assumed to be five years or 43,800 hours. For this large MTTF to make some sense, disk manufacturers argue that the model corresponds to a user who buys a disk and then keeps replacing the disk every 5 years—the planned lifetime of the disk. The claim is that if many customers (and their great-grandchildren) did this for the next century, on average they would replace a disk 27 times before a failure, or about 140 years.

A more useful measure is the percentage of disks that fail, which is called the *annual failure rate*. Assume 1000 disks with a 1,000,000-hour MTTF and that the disks are used 24 hours a day. If you replaced failed disks with a new one having the same reliability characteristics, the number that would fail in a year (8760 hours) is

$$\text{Failed disks} = \frac{\text{Number of disks} \times \text{Time period}}{\text{MTTF}} = \frac{1000\ \text{disks} \times 8760\ \text{hours/drive}}{1{,}000{,}000\ \text{hours/failure}} = 9$$

Stated alternatively, 0.9% would fail per year, or 4.4% over a 5-year lifetime.

Moreover, those high numbers are quoted assuming limited ranges of temperature and vibration; if they are exceeded, then all bets are off. A survey of disk drives in real environments (Gray and van Ingen, 2005) found that 3%–7% of drives failed per year, for an MTTF of about 125,000–300,000 hours. An even larger study found annual disk failure rates of 2%–10% (Pinheiro et al., 2007). Therefore the real-world MTTF is about 2–10 times worse than the manufacturer's MTTF.

Fallacy *Peak performance tracks observed performance.*

The only universally true definition of peak performance is "the performance level a computer is guaranteed not to exceed." Figure 1.24 shows the percentage of peak performance for four programs on four multiprocessors. It varies from 5% to 58%. Since the gap is so large and can vary significantly by benchmark, peak performance is not generally useful in predicting observed performance.

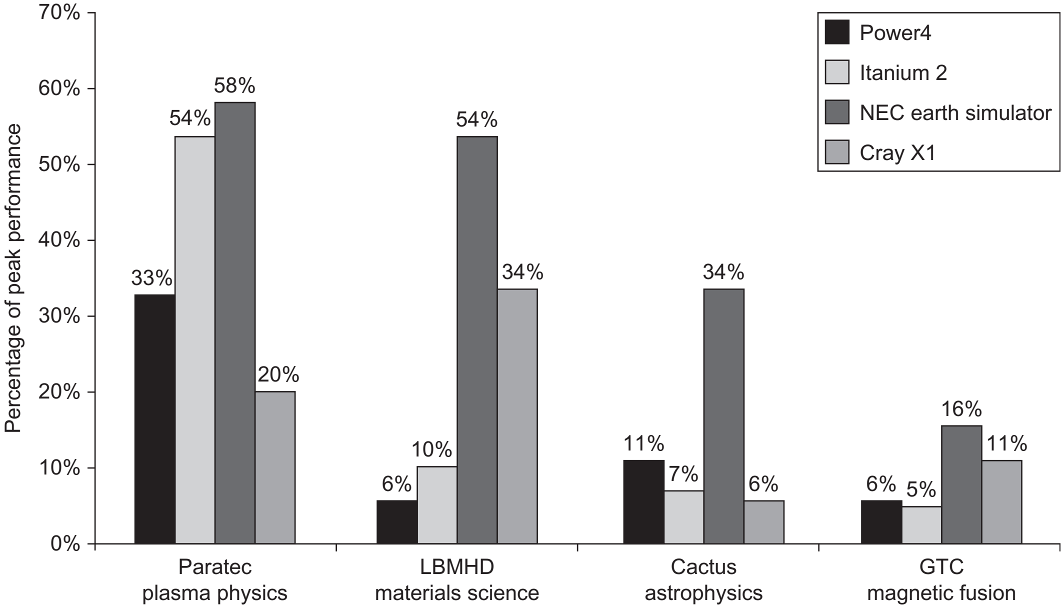


Figure 1.24 Percentage of peak performance for four programs on four multiprocessors scaled to 64 processors. The Earth Simulator and X1 are vector processors (see Chapter 4 and Appendix G). Not only did they deliver a higher fraction of peak performance, but they also had the highest peak performance and the lowest clock rates. Except for the Paratec program, the Power 4 and Itanium 2 systems delivered between 5% and 10% of their peak. From Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S., 2004. Scientific computations on modern parallel vector systems. In: Proc. ACM/IEEE Conf. on Supercomputing, November 6–12, 2004, Pittsburgh, Penn., p. 10.

Pitfall *Fault detection can lower availability.*

This apparently ironic pitfall is because computer hardware has a fair amount of state that may not always be critical to proper operation. For example, it is not fatal if an error occurs in a branch predictor, because only performance may suffer.

In processors that try to exploit ILP aggressively, not all the operations are needed for correct execution of the program. Mukherjee et al. (2003) found that less than 30% of the operations were potentially on the critical path for the SPEC2000 benchmarks.

The same observation is true about programs. If a register is "dead" in a program—that is, the program will write the register before it is read again—then errors do not matter. If you were to crash the program upon detection of a transient fault in a dead register, it would lower availability unnecessarily.

The Sun Microsystems Division of Oracle lived this pitfall in 2000 with an L2 cache that included parity, but not error correction, in its Sun E3000 to Sun E10000 systems. The SRAMs they used to build the caches had intermittent faults, which parity detected. If the data in the cache were not modified, the processor would simply reread the data from the cache. Because the designers did not protect the cache with ECC (error-correcting code), the operating system had no choice but to report an error to dirty data and crash the program. Field engineers found no problems on inspection in more than 90% of the cases.

To reduce the frequency of such errors, Sun modified the Solaris operating system to "scrub" the cache by having a process that proactively wrote dirty data to memory. Because the processor chips did not have enough pins to add ECC, the only hardware option for dirty data was to duplicate the external cache, using the copy without the parity error to correct the error.

The pitfall is in detecting faults without providing a mechanism to correct them. These engineers are unlikely to design another computer without ECC on external caches.

1.12 Concluding Remarks

This chapter has introduced a number of concepts and provided a quantitative framework that we will expand on throughout the book. Starting with the last edition, energy efficiency is the constant companion to performance.

In Chapter 2, we start with the all-important area of memory system design. We will examine a wide range of techniques that conspire to make memory look infinitely large while still being as fast as possible. (Appendix B provides introductory material on caches for readers without much experience and background with them.) As in later chapters, we will see that hardware-software cooperation has become a key to high-performance memory systems, just as it has to high-performance pipelines. This chapter also covers virtual machines, an increasingly important technique for protection.

In Chapter 3, we look at ILP, of which pipelining is the simplest and most common form. Exploiting ILP is one of the most important techniques for building high-speed uniprocessors. Chapter 3 begins with an extensive discussion of basic concepts that will prepare you for the wide range of ideas examined in both chapters. Chapter 3 uses examples that span about 40 years, drawing from one of the first supercomputers (IBM 360/91) to the fastest processors on the market in 2017. It emphasizes what is called the *dynamic* or *runtime approach* to exploiting ILP. It also talks about the limits to ILP ideas and introduces multithreading, which is further developed in both Chapters 4 and 5. Appendix C provides introductory material on pipelining for readers without much experience and background in pipelining. (We expect it to be a review for many readers, including those of our introductory text, *Computer Organization and Design: The Hardware/Software Interface*.)

Chapter 4 explains three ways to exploit data-level parallelism. The classic and oldest approach is vector architecture, and we start there to lay down the principles of SIMD design. (Appendix G goes into greater depth on vector architectures.) We next explain the SIMD instruction set extensions found in most desktop microprocessors today. The third piece is an in-depth explanation of how modern graphics processing units (GPUs) work. Most GPU descriptions are written from the programmer's perspective, which usually hides how the computer really works. This section explains GPUs from an insider's perspective, including a mapping between GPU jargon and more traditional architecture terms.

Chapter 5 focuses on the issue of achieving higher performance using multiple processors, or multiprocessors. Instead of using parallelism to overlap individual instructions, multiprocessing uses parallelism to allow multiple instruction streams to be executed simultaneously on different processors. Our focus is on the dominant form of multiprocessors, shared-memory multiprocessors, though we introduce other types as well and discuss the broad issues that arise in any multiprocessor. Here again we explore a variety of techniques, focusing on the important ideas first introduced in the 1980s and 1990s.

Chapter 6 introduces clusters and then goes into depth on WSCs, which computer architects help design. The designers of WSCs are the professional descendants of the pioneers of supercomputers, such as Seymour Cray, in that they are designing extreme computers. WSCs contain tens of thousands of servers, and the equipment and the building that holds them cost nearly $200 million. The concerns of price-performance and energy efficiency of the earlier chapters apply to WSCs, as does the quantitative approach to making decisions.

Chapter 7 is new to this edition. It introduces domain-specific architectures as the only path forward for improved performance and energy efficiency given the end of Moore's Law and Dennard scaling. It offers guidelines on how to build effective domain-specific architectures, introduces the exciting domain of deep neural networks, describes four recent examples that take very different approaches to accelerating neural networks, and then compares their cost-performance.

This book comes with an abundance of material online (see Preface for more details), both to reduce cost and to introduce readers to a variety of advanced topics. Figure 1.25 shows them all. Appendices A–C, which appear in the book, will be a review for many readers.

Appendix	Title
A	Instruction Set Principles
B	Review of Memory Hierarchies
C	Pipelining: Basic and Intermediate Concepts
D	Storage Systems
E	Embedded Systems
F	Interconnection Networks
G	Vector Processors in More Depth
H	Hardware and Software for VLIW and EPIC
I	Large-Scale Multiprocessors and Scientific Applications
J	Computer Arithmetic
K	Survey of Instruction Set Architectures
L	Advanced Concepts on Address Translation
M	Historical Perspectives and References

Figure 1.25 List of appendices.

In Appendix D, we move away from a processor-centric view and discuss issues in storage systems. We apply a similar quantitative approach, but one based on observations of system behavior and using an end-to-end approach to performance analysis. This appendix addresses the important issue of how to store and retrieve data efficiently using primarily lower-cost magnetic storage technologies. Our focus is on examining the performance of disk storage systems for typical I/O-intensive workloads, such as the OLTP benchmarks mentioned in this chapter. We extensively explore advanced topics in RAID-based systems, which use redundant disks to achieve both high performance and high availability. Finally, Appendix D introduces queuing theory, which gives a basis for trading off utilization and latency.

Appendix E applies an embedded computing perspective to the ideas of each of the chapters and early appendices.

Appendix F explores the topic of system interconnect broadly, including wide area and system area networks that allow computers to communicate.

Appendix H reviews VLIW hardware and software, which, in contrast, are less popular than when EPIC appeared on the scene just before the last edition.

Appendix I describes large-scale multiprocessors for use in high-performance computing.

Appendix J is the only appendix that remains from the first edition, and it covers computer arithmetic.

Appendix K provides a survey of instruction architectures, including the 80x86, the IBM 360, the VAX, and many RISC architectures, including ARM, MIPS, Power, RISC-V, and SPARC.

Appendix L is new and discusses advanced techniques for memory management, focusing on support for virtual machines and design of address translation for very large address spaces. With the growth in cloud processors, these architectural enhancements are becoming more important.

We describe Appendix M next.

1.13 Historical Perspectives and References

Appendix M (available online) includes historical perspectives on the key ideas presented in each of the chapters in this text. These historical perspective sections allow us to trace the development of an idea through a series of machines or to describe significant projects. If you're interested in examining the initial development of an idea or processor or want further reading, references are provided at the end of each history. For this chapter, see Section M.2, "The Early Development of Computers," for a discussion on the early development of digital computers and performance measurement methodologies.

As you read the historical material, you'll soon come to realize that one of the important benefits of the youth of computing, compared to many other engineering fields, is that some of the pioneers are still alive—we can learn the history by simply asking them!

Case Studies and Exercises by Diana Franklin

Case Study 1: Chip Fabrication Cost

Concepts illustrated by this case study

- Fabrication Cost
- Fabrication Yield
- Defect Tolerance Through Redundancy

Many factors are involved in the price of a computer chip. Intel is spending $7 billion to complete its Fab 42 fabrication facility for 7 nm technology. In this case study, we explore a hypothetical company in the same situation and how different design decisions involving fabrication technology, area, and redundancy affect the cost of chips.

1.1 [10/10] <1.6> Figure 1.26 gives hypothetical relevant chip statistics that influence the cost of several current chips. In the next few exercises, you will be exploring the effect of different possible design decisions for the Intel chips.

Chip	Die Size (mm^2)	Estimated defect rate (per cm^2)	*N*	Manufacturing size (nm)	Transistors (billion)	Cores
BlueDragon	180	0.03	12	10	7.5	4
RedDragon	120	0.04	14	7	7.5	4
Phoenix[8]	200	0.04	14	7	12	8

Figure 1.26 Manufacturing cost factors for several hypothetical current and future processors.

a. [10] <1.6> What is the yield for the Phoenix chip?

b. [10] <1.6> Why does Phoenix have a higher defect rate than BlueDragon?

1.2 [20/20/20/20] <1.6> They will sell a range of chips from that factory, and they need to decide how much capacity to dedicate to each chip. Imagine that they will sell two chips. Phoenix is a completely new architecture designed with 7 nm technology in mind, whereas RedDragon is the same architecture as their 10 nm BlueDragon. Imagine that RedDragon will make a profit of $15 per defect-free chip. Phoenix will make a profit of $30 per defect-free chip. Each wafer has a 450 mm diameter.

a. [20] <1.6> How much profit do you make on each wafer of Phoenix chips?

b. [20] <1.6> How much profit do you make on each wafer of RedDragon chips?

c. [20] <1.6> If your demand is 50,000 RedDragon chips per month and 25,000 Phoenix chips per month, and your facility can fabricate 70 wafers a month, how many wafers should you make of each chip?

1.3 [20/20] <1.6> Your colleague at AMD suggests that, since the yield is so poor, you might make chips more cheaply if you released multiple versions of the same chip, just with different numbers of cores. For example, you could sell Phoenix^8, Phoenix^4, Phoenix^2, and Phoenix^1, which contain 8, 4, 2, and 1 cores on each chip, respectively. If all eight cores are defect-free, then it is sold as Phoenix^8. Chips with four to seven defect-free cores are sold as Phoenix^4, and those with two or three defect-free cores are sold as Phoenix^2. For simplification, calculate the yield for a single core as the yield for a chip that is 1/8 the area of the original Phoenix chip. Then view that yield as an independent probability of a single core being defect free. Calculate the yield for each configuration as the probability of at the corresponding number of cores being defect free.

a. [20] <1.6> What is the yield for a single core being defect free as well as the yield for Phoenix^4, Phoenix^2 and Phoenix^1?

b. [5] <1.6> Using your results from part a, determine which chips you think it would be worthwhile to package and sell, and why.

c. [10] <1.6> If it previously cost $20 dollars per chip to produce Phoenix^8, what will be the cost of the new Phoenix chips, assuming that there are no additional costs associated with rescuing them from the trash?

d. [20] <1.6> You currently make a profit of $30 for each defect-free Phoenix^8, and you will sell each Phoenix^4 chip for $25. How much is your profit per Phoenix^8 chip if you consider (i) the purchase price of Phoenix^4 chips to be entirely profit and (ii) apply the profit of Phoenix^4 chips to each Phoenix^8 chip in proportion to how many are produced? Use the yields calculated from part Problem 1.3a, not from problem 1.1a.

Case Study 2: Power Consumption in Computer Systems

Concepts illustrated by this case study

- Amdahl's Law
- Redundancy
- MTTF
- Power Consumption

Power consumption in modern systems is dependent on a variety of factors, including the chip clock frequency, efficiency, and voltage. The following exercises explore the impact on power and energy that different design decisions and use scenarios have.

1.4 [10/10/10/10] <1.5> A cell phone performs very different tasks, including streaming music, streaming video, and reading email. These tasks perform very different computing tasks. Battery life and overheating are two common problems for cell phones, so reducing power and energy consumption are critical. In this problem, we consider what to do when the user is not using the phone to its full computing capacity. For these problems, we will evaluate an unrealistic scenario in which the cell phone has no specialized processing units. Instead, it has a quad-core, general-purpose processing unit. Each core uses 0.5 W at full use. For email-related tasks, the quad-core is 8× as fast as necessary.

a. [10] <1.5> How much dynamic energy and power are required compared to running at full power? First, suppose that the quad-core operates for 1/8 of the time and is idle for the rest of the time. That is, the clock is disabled for 7/8 of the time, with no leakage occurring during that time. Compare total dynamic energy as well as dynamic power while the core is running.

b. [10] <1.5> How much dynamic energy and power are required using frequency and voltage scaling? Assume frequency and voltage are both reduced to 1/8 the entire time.

c. [10] <1.6, 1.9> Now assume the voltage may not decrease below 50% of the original voltage. This voltage is referred to as the *voltage floor*, and any voltage lower than that will lose the state. Therefore, while the frequency can keep decreasing, the voltage cannot. What are the dynamic energy and power savings in this case?

d. [10] <1.5> How much energy is used with a dark silicon approach? This involves creating specialized ASIC hardware for each major task and power gating those elements when not in use. Only one general-purpose core would be provided, and the rest of the chip would be filled with specialized units. For email, the one core would operate for 25% the time and be turned completely off with power gating for the other 75% of the time. During the other 75% of the time, a specialized ASIC unit that requires 20% of the energy of a core would be running.

1.5 [10/10/10] <1.5> As mentioned in Exercise 1.4, cell phones run a wide variety of applications. We'll make the same assumptions for this exercise as the previous one, that it is 0.5 W per core and that a quad core runs email 3× as fast.

a. [10] <1.5> Imagine that 80% of the code is parallelizable. By how much would the frequency and voltage on a single core need to be increased in order to execute at the same speed as the four-way parallelized code?

b. [10] <1.5> What is the reduction in dynamic energy from using frequency and voltage scaling in part a?

c. [10] <1.5> How much energy is used with a dark silicon approach? In this approach, all hardware units are power gated, allowing them to turn off entirely (causing no leakage). Specialized ASICs are provided that perform the same computation for 20% of the power as the general-purpose processor. Imagine that each core is power gated. The video game requires two ASICS and two cores. How much dynamic energy does it require compared to the baseline of parallelized on four cores?

1.6 [10/10/10/10/10/20] <1.5,1.9> General-purpose processes are optimized for general-purpose computing. That is, they are optimized for behavior that is generally found across a large number of applications. However, once the domain is restricted somewhat, the behavior that is found across a large number of the target applications may be different from general-purpose applications. One such application is deep learning or neural networks. Deep learning can be applied to many different applications, but the fundamental building block of inference—using the learned information to make decisions—is the same across them all. Inference operations are largely parallel, so they are currently performed on graphics processing units, which are specialized more toward this type of computation, and not to inference in particular. In a quest for more performance per watt, Google has created a custom chip using tensor processing units to accelerate inference operations in deep learning.[1] This approach can be used for speech recognition and image recognition, for example. This problem explores the trade-offs between this process, a general-purpose processor (Haswell E5-2699 v3) and a GPU (NVIDIA K80), in terms of performance and cooling. If heat is not removed from the computer efficiently, the fans will blow hot air back onto the computer, not cold air. Note: The differences are more than processor—on-chip memory and DRAM also come into play. Therefore statistics are at a system level, not a chip level.

---

[1] Cite paper at this website: https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view.

a. [10] <1.9> If Google's data center spends 70% of its time on workload A and 30% of its time on workload B when running GPUs, what is the speedup of the TPU system over the GPU system?

b. [10] <1.9> If Google's data center spends 70% of its time on workload A and 30% of its time on workload B when running GPUs, what percentage of Max IPS does it achieve for each of the three systems?

c. [15] <1.5, 1.9> Building on (b), assuming that the power scales linearly from idle to busy power as IPS grows from 0% to 100%, what is the performance per watt of the TPU system over the GPU system?

d. [10] <1.9> If another data center spends 40% of its time on workload A, 10% of its time on workload B, and 50% of its time on workload C, what are the speedups of the GPU and TPU systems over the general-purpose system?

e. [10] <1.5> A cooling door for a rack costs $4000 and dissipates 14 kW (into the room; additional cost is required to get it out of the room). How many Haswell-, NVIDIA-, or Tensor-based servers can you cool with one cooling door, assuming TDP in Figures 1.27 and 1.28?

f. [20] <1.5> Typical server farms can dissipate a maximum of 200 W per square foot. Given that a server rack requires 11 square feet (including front and back clearance), how many servers from part (e) can be placed on a single rack, and how many cooling doors are required?

System	Chip	TDP	Idle power	Busy power
General-purpose	Haswell E5-2699 v3	504 W	159 W	455 W
Graphics processor	NVIDIA K80	1838 W	357 W	991 W
Custom ASIC	TPU	861 W	290 W	384 W

Figure 1.27 Hardware characteristics for general-purpose processor, graphical processing unit-based or custom ASIC-based system, including measured power (cite ISCA paper).

System	Chip	Throughput			% Max IPS		
		A	B	C	A	B	C
General-purpose	Haswell E5-2699 v3	5482	13,194	12,000	42%	100%	90%
Graphics processor	NVIDIA K80	13,461	36,465	15,000	37%	100%	40%
Custom ASIC	TPU	225,000	280,000	2000	80%	100%	1%

Figure 1.28 Performance characteristics for general-purpose processor, graphical processing unit-based or custom ASIC-based system on two neural-net workloads (cite ISCA paper). Workloads A and B are from published results. Workload C is a fictional, more general-purpose application.

Exercises

1.7 [10/15/15/10/10] <1.4, 1.5> One challenge for architects is that the design created today will require several years of implementation, verification, and testing before appearing on the market. This means that the architect must project what the technology will be like several years in advance. Sometimes, this is difficult to do.

a. [10] <1.4> According to the trend in device scaling historically observed by Moore's Law, the number of transistors on a chip in 2025 should be how many times the number in 2015?

b. [15] <1.5> The increase in performance once mirrored this trend. Had performance continued to climb at the same rate as in the 1990s, approximately what performance would chips have over the VAX-11/780 in 2025?

c. [15] <1.5> At the current rate of increase of the mid-2000s, what is a more updated projection of performance in 2025?

d. [10] <1.4> What has limited the rate of growth of the clock rate, and what are architects doing with the extra transistors now to increase performance?

e. [10] <1.4> The rate of growth for DRAM capacity has also slowed down. For 20 years, DRAM capacity improved by 60% each year. If 8 Gbit DRAM was first available in 2015, and 16 Gbit is not available until 2019, what is the current DRAM growth rate?

1.8 [10/10] <1.5> You are designing a system for a real-time application in which specific deadlines must be met. Finishing the computation faster gains nothing. You find that your system can execute the necessary code, in the worst case, twice as fast as necessary.

a. [10] <1.5> How much energy do you save if you execute at the current speed and turn off the system when the computation is complete?

b. [10] <1.5> How much energy do you save if you set the voltage and frequency to be half as much?

1.9 [10/10/20/20] <1.5> Server farms such as Google and Yahoo! provide enough compute capacity for the highest request rate of the day. Imagine that most of the time these servers operate at only 60% capacity. Assume further that the power does not scale linearly with the load; that is, when the servers are operating at 60% capacity, they consume 90% of maximum power. The servers could be turned off, but they would take too long to restart in response to more load. A new system has been proposed that allows for a quick restart but requires 20% of the maximum power while in this "barely alive" state.

a. [10] <1.5> How much power savings would be achieved by turning off 60% of the servers?

b. [10] <1.5> How much power savings would be achieved by placing 60% of the servers in the "barely alive" state?

c. [20] <1.5> How much power savings would be achieved by reducing the voltage by 20% and frequency by 40%?

d. [20] <1.5> How much power savings would be achieved by placing 30% of the servers in the "barely alive" state and 30% off?

1.10 [10/10/20] <1.7> Availability is the most important consideration for designing servers, followed closely by scalability and throughput.

a. [10] <1.7> We have a single processor with a failure in time (FIT) of 100. What is the mean time to failure (MTTF) for this system?

b. [10] <1.7> If it takes one day to get the system running again, what is the availability of the system?

c. [20] <1.7> Imagine that the government, to cut costs, is going to build a supercomputer out of inexpensive computers rather than expensive, reliable computers. What is the MTTF for a system with 1000 processors? Assume that if one fails, they all fail.

1.11 [20/20/20] <1.1, 1.2, 1.7> In a server farm such as that used by Amazon or eBay, a single failure does not cause the entire system to crash. Instead, it will reduce the number of requests that can be satisfied at any one time.

a. [20] <1.7> If a company has 10,000 computers, each with an MTTF of 35 days, and it experiences catastrophic failure only if 1/3 of the computers fail, what is the MTTF for the system?

b. [20] <1.1, 1.7> If it costs an extra $1000, per computer, to double the MTTF, would this be a good business decision? Show your work.

c. [20] <1.2> Figure 1.3 shows, on average, the cost of downtimes, assuming that the cost is equal at all times of the year. For retailers, however, the Christmas season is the most profitable (and therefore the most costly time to lose sales). If a catalog sales center has twice as much traffic in the fourth quarter as every other quarter, what is the average cost of downtime per hour during the fourth quarter and the rest of the year?

1.12 [20/10/10/10/15] <1.9> In this exercise, assume that we are considering enhancing a quad-core machine by adding encryption hardware to it. When computing encryption operations, it is 20 times faster than the normal mode of execution. We will define percentage of encryption as the percentage of time in the original execution that is spent performing encryption operations. The specialized hardware increases power consumption by 2%.

a. [20] <1.9> Draw a graph that plots the speedup as a percentage of the computation spent performing encryption. Label the y-axis "Net speedup" and label the x-axis "Percent encryption."

b. [10] <1.9> With what percentage of encryption will adding encryption hardware result in a speedup of 2?

c. [10] <1.9> What percentage of time in the new execution will be spent on encryption operations if a speedup of 2 is achieved?

d. [15] <1.9> Suppose you have measured the percentage of encryption to be 50%. The hardware design group estimates it can speed up the encryption hardware even more with significant additional investment. You wonder whether adding a second unit in order to support parallel encryption operations would be more useful. Imagine that in the original program, 90% of the encryption operations could be performed in parallel. What is the speedup of providing two or four encryption units, assuming that the parallelization allowed is limited to the number of encryption units?

1.13 [15/10] <1.9> Assume that we make an enhancement to a computer that improves some mode of execution by a factor of 10. Enhanced mode is used 50% of the time, measured as a percentage of the execution time *when the enhanced mode is in use*. Recall that Amdahl's Law depends on the fraction of the original, unenhanced execution time that could make use of enhanced mode. Thus we cannot directly use this 50% measurement to compute speedup with Amdahl's Law.

a. [15] <1.9> What is the speedup we have obtained from fast mode?

b. [10] <1.9> What percentage of the original execution time has been converted to fast mode?

1.14 [20/20/15] <1.9> When making changes to optimize part of a processor, it is often the case that speeding up one type of instruction comes at the cost of slowing down something else. For example, if we put in a complicated fast floating-point unit, that takes space, and something might have to be moved farther away from the middle to accommodate it, adding an extra cycle in delay to reach that unit. The basic Amdahl's Law equation does not take into account this trade-off.

a. [20] <1.9> If the new fast floating-point unit speeds up floating-point operations by, on average, 2x, and floating-point operations take 20% of the original program's execution time, what is the overall speedup (ignoring the penalty to any other instructions)?

b. [20] <1.9> Now assume that speeding up the floating-point unit slowed down data cache accesses, resulting in a 1.5x slowdown (or 2/3 speedup). Data cache accesses consume 10% of the execution time. What is the overall speedup now?

c. [15] <1.9> After implementing the new floating-point operations, what percentage of execution time is spent on floating-point operations? What percentage is spent on data cache accesses?

1.15 [10/10/20/20] <1.10> Your company has just bought a new 22-core processor, and you have been tasked with optimizing your software for this processor. You will run four applications on this system, but the resource requirements are not equal. Assume the system and application characteristics listed in Table 1.1.

Table 1.1 Four applications

Application	A	B	C	D
% resources needed	41	27	18	14
% parallelizable	50	80	60	90

The percentage of resources of assuming they are all run in serial. Assume that when you parallelize a portion of the program by X, the speedup for that portion is X.

a. [10] <1.10> How much speedup would result from running application A on the entire 22-core processor, as compared to running it serially?

b. [10] <1.10> How much speedup would result from running application D on the entire 22-core processor, as compared to running it serially?

c. [20] <1.10> Given that application A requires 41% of the resources, if we statically assign it 41% of the cores, what is the overall speedup if A is run parallelized but everything else is run serially?

d. [20] <1.10> What is the overall speedup if all four applications are statically assigned some of the cores, relative to their percentage of resource needs, and all run parallelized?

e. [10] <1.10> Given acceleration through parallelization, what new percentage of the resources are the applications receiving, considering only active time on their statically-assigned cores?

1.16 [10/20/20/20/25] <1.10> When parallelizing an application, the ideal speedup is speeding up by the number of processors. This is limited by two things: percentage of the application that can be parallelized and the cost of communication. Amdahl's Law takes into account the former but not the latter.

a. [10] <1.10> What is the speedup with N processors if 80% of the application is parallelizable, ignoring the cost of communication?

b. [20] <1.10> What is the speedup with eight processors if, for every processor added, the communication overhead is 0.5% of the original execution time.

c. [20] <1.10> What is the speedup with eight processors if, for every time the number of processors is doubled, the communication overhead is increased by 0.5% of the original execution time?

d. [20] <1.10> What is the speedup with N processors if, for every time the number of processors is doubled, the communication overhead is increased by 0.5% of the original execution time?

e. [25] <1.10> Write the general equation that solves this question: What is the number of processors with the highest speedup in an application in which P% of the original execution time is parallelizable, and, for every time the number of processors is doubled, the communication is increased by 0.5% of the original execution time?

2.1 Introduction 78
2.2 Memory Technology and Optimizations 84
2.3 Ten Advanced Optimizations of Cache Performance 94
2.4 Virtual Memory and Virtual Machines 118
2.5 Cross-Cutting Issues: The Design of Memory Hierarchies 126
2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700 129
2.7 Fallacies and Pitfalls 142
2.8 Concluding Remarks: Looking Ahead 146
2.9 Historical Perspectives and References 148
Case Studies and Exercises by Norman P. Jouppi, Rajeev Balasubramonian, Naveen Muralimanohar, and Sheng Li 148

2

Memory Hierarchy Design

> Ideally one would desire an indefinitely large memory capacity such that any particular… word would be immediately available… We are… forced to recognize the possibility of constructing a hierarchy of memories each of which has greater capacity than the preceding but which is less quickly accessible.
>
> A. W. Burks, H. H. Goldstine,
> and J. von Neumann,
> *Preliminary Discussion of the*
> *Logical Design of an Electronic*
> *Computing Instrument* (1946).

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00002-X

© 2019 Elsevier Inc. All rights reserved.

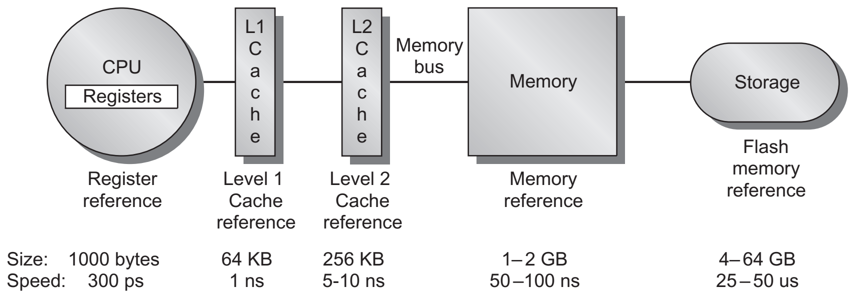
2.1 Introduction

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. An economical solution to that desire is a memory hierarchy, which takes advantage of locality and trade-offs in the cost-performance of memory technologies. The principle of locality, presented in the first chapter, says that most programs do not access all code or data uniformly. Locality occurs in time (temporal locality) and in space (spatial locality). This principle plus the guideline that for a given implementation technology and power budget, smaller hardware can be made faster led to hierarchies based on memories of different speeds and sizes. Figure 2.1 shows several different multilevel memory hierarchies, including typical sizes and speeds of access. As Flash and next generation memory technologies continue to close the gap with disks in cost per bit, such technologies are likely to increasingly replace magnetic disks for secondary storage. As Figure 2.1 shows, these technologies are already used in many personal computers and increasingly in servers, where the advantages in performance, power, and density are significant.

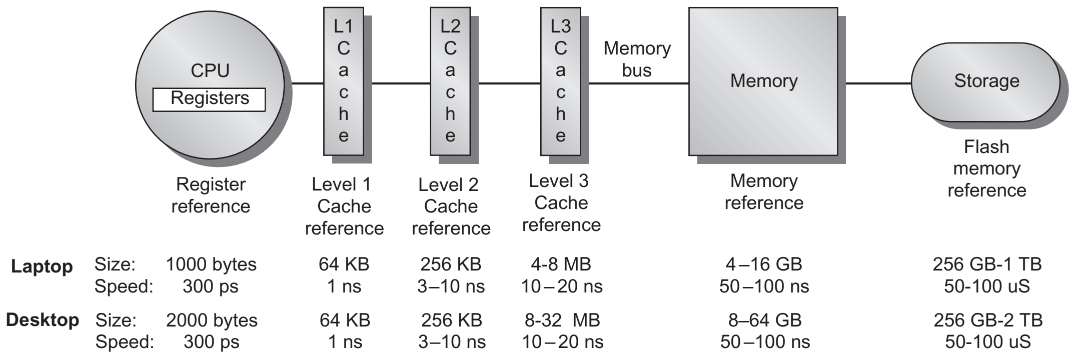
Because fast memory is more expensive, a memory hierarchy is organized into several levels—each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. The goal is to provide a memory system with a cost per byte that is almost as low as the cheapest level of memory and a speed almost as fast as the fastest level. In most cases (but not all), the data contained in a lower level are a superset of the next higher level. This property, called the *inclusion property*, is always required for the lowest level of the hierarchy, which consists of main memory in the case of caches and secondary storage (disk or Flash) in the case of virtual memory.

The importance of the memory hierarchy has increased with advances in performance of processors. Figure 2.2 plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency), assuming a single DRAM and a single memory bank. The reality is more complex because the processor request rate is not uniform, and the memory system typically has multiple banks of DRAMs and channels. Although the gap in access time increased significantly for many years, the lack of significant performance improvement in single processors has led to a slowdown in the growth of the gap between processors and DRAM.

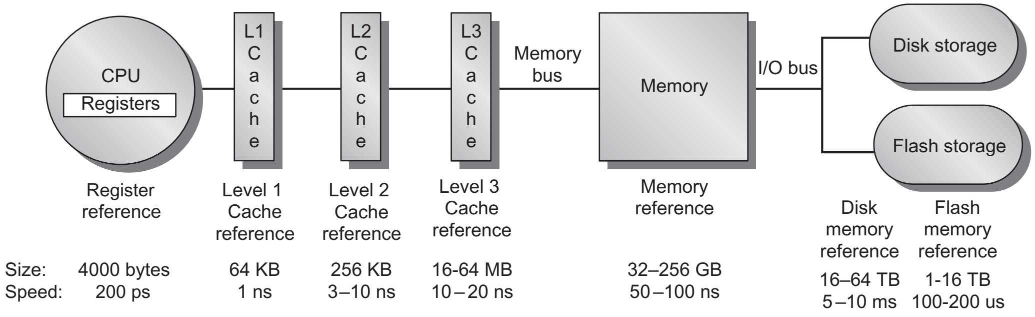
Because high-end processors have multiple cores, the bandwidth requirements are greater than for single cores. Although single-core bandwidth has grown more slowly in recent years, the gap between CPU memory demand and DRAM bandwidth continues to grow as the numbers of cores grow. A modern high-end desktop processor such as the Intel Core i7 6700 can generate two data memory references per core each clock cycle. With four cores and a 4.2 GHz clock rate, the i7 can generate a peak of 32.8 billion 64-bit data memory references per second, in addition to a peak instruction demand of about 12.8 billion 128-bit instruction


CPU
Registers
L1 Cache
L2 Cache
Memory bus
Memory
Storage
Register reference
Level 1 Cache reference
Level 2 Cache reference
Memory reference
Flash memory reference
Size: 1000 bytes
Speed: 300 ps
64 KB
1 ns
256 KB
5-10 ns
1–2 GB
50–100 ns
4–64 GB
25–50 us

(A) Memory hierarchy for a personal mobile device


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Memory bus
Memory
Storage
Register reference
Level 1 Cache reference
Level 2 Cache reference
Level 3 Cache reference
Memory reference
Flash memory reference
Laptop
Size: 1000 bytes
Speed: 300 ps
64 KB
1 ns
256 KB
3–10 ns
4-8 MB
10–20 ns
4–16 GB
50–100 ns
256 GB-1 TB
50-100 uS
Desktop
Size: 2000 bytes
Speed: 300 ps
64 KB
1 ns
256 KB
3–10 ns
8-32 MB
10–20 ns
8–64 GB
50–100 ns
256 GB-2 TB
50-100 uS

(B) Memory hierarchy for a laptop or a desktop


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Memory bus
Memory
I/O bus
Disk storage
Flash storage
Register reference
Level 1 Cache reference
Level 2 Cache reference
Level 3 Cache reference
Memory reference
Disk memory reference
Flash memory reference
Size: 4000 bytes
Speed: 200 ps
64 KB
1 ns
256 KB
3–10 ns
16-64 MB
10–20 ns
32–256 GB
50–100 ns
16–64 TB
5–10 ms
1-16 TB
100-200 us

(C) Memory hierarchy for server

Figure 2.1 The levels in a typical memory hierarchy in a personal mobile device (PMD), such as a cell phone or tablet (A), in a laptop or desktop computer (B), and in a server (C). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9 from picoseconds to milliseconds in the case of magnetic disks and that the size units change by a factor of 10^{10} from thousands of bytes to tens of terabytes. If we were to add warehouse-sized computers, as opposed to just servers, the capacity scale would increase by three to six orders of magnitude. Solid-state drives (SSDs) composed of Flash are used exclusively in PMDs, and heavily in both laptops and desktops. In many desktops, the primary storage system is SSD, and expansion disks are primarily hard disk drives (HDDs). Likewise, many servers mix SSDs and HDDs.

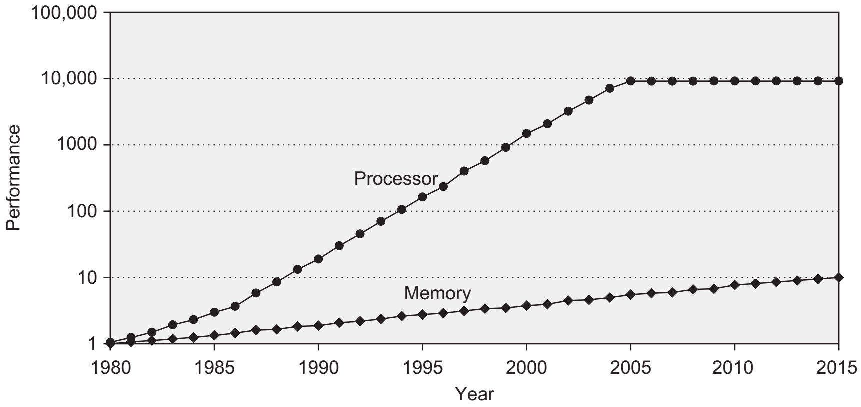


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. In mid-2017, AMD, Intel and Nvidia all announced chip sets using versions of HBM technology. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KiB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.4 on page 88). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and only small improvements in processor performance (on a per-core basis) between 2005 and 2015. As you can see, until 2010 memory access times in DRAM improved slowly but consistently; since 2010 the improvement in access time has reduced, as compared with the earlier periods, although there have been continued improvements in bandwidth. See Figure 1.1 in Chapter 1 for more information.

references; this is a total peak demand bandwidth of 409.6 GiB/s! This incredible bandwidth is achieved by multiporting and pipelining the caches; by using three levels of caches, with two private levels per core and a shared L3; and by using a separate instruction and data cache at the first level. In contrast, the peak bandwidth for DRAM main memory, using two memory channels, is only 8% of the demand bandwidth (34.1 GiB/s). Upcoming versions are expected to have an L4 DRAM cache using embedded or stacked DRAM (see Sections 2.2 and 2.3).

Traditionally, designers of memory hierarchies focused on optimizing average memory access time, which is determined by the cache access time, miss rate, and miss penalty. More recently, however, power has become a major consideration. In high-end microprocessors, there may be 60 MiB or more of on-chip cache, and a large second- or third-level cache will consume significant power both as leakage when not operating (called *static power*) and as active power, as when performing a read or write (called *dynamic power*), as described in Section 2.3. The problem is even more acute in processors in PMDs where the CPU is less aggressive and the power budget may be 20 to 50 times smaller. In such cases, the caches can account for 25% to 50% of the total power consumption. Thus more designs must consider both performance and power trade-offs, and we will examine both in this chapter.

Basics of Memory Hierarchies: A Quick Review

The increasing size and thus importance of this gap led to the migration of the basics of memory hierarchy into undergraduate courses in computer architecture, and even to courses in operating systems and compilers. Thus we'll start with a quick review of caches and their operation. The bulk of the chapter, however, describes more advanced innovations that attack the processor—memory performance gap.

When a word is not found in the cache, the word must be fetched from a lower level in the hierarchy (which may be another cache or the main memory) and placed in the cache before continuing. Multiple words, called a *block* (or *line*), are moved for efficiency reasons, and because they are likely to be needed soon due to spatial locality. Each cache block includes a *tag* to indicate which memory address it corresponds to.

A key design decision is where blocks (or lines) can be placed in a cache. The most popular scheme is *set associative*, where a *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. Finding a block consists of first mapping the block address to the set and then searching the set—usually in parallel—to find the block. The set is chosen by the address of the data:

$$(\textit{Block address}) \text{ MOD } (\textit{Number of sets in cache})$$

If there are n blocks in a set, the cache placement is called *n-way set associative*. The end points of set associativity have their own names. A *direct-mapped* cache has just one block per set (so a block is always placed in the same location), and a *fully associative* cache has just one set (so a block can be placed anywhere).

Caching data that is only read is easy because the copy in the cache and memory will be identical. Caching writes is more difficult; for example, how can the copy in the cache and memory be kept consistent? There are two main strategies. A *write-through* cache updates the item in the cache *and* writes through to update main memory. A *write-back* cache only updates the copy in the cache. When the block is about to be replaced, it is copied back to memory. Both write strategies can use a *write buffer* to allow the cache to proceed as soon as the data are placed in the buffer rather than wait for full latency to write the data into memory.

One measure of the benefits of different cache organizations is miss rate. *Miss rate* is simply the fraction of cache accesses that result in a miss—that is, the number of accesses that miss divided by the number of accesses.

To gain insights into the causes of high miss rates, which can inspire better cache designs, the three Cs model sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. Compulsory misses are those that occur even if you were to have an infinite-sized cache.
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.

- *Conflict*—If the block placement strategy is not fully associative, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if multiple blocks map to its set and accesses to the different blocks are intermingled.

Figure B.8 on page 24 shows the relative frequency of cache misses broken down by the three Cs. As mentioned in Appendix B, the three C's model is conceptual, and although its insights usually hold, it is not a definitive model for explaining the cache behavior of individual references.

As we will see in Chapters 3 and 5, multithreading and multiple cores add complications for caches, both increasing the potential for capacity misses as well as adding a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we will consider these issues in Chapter 5.

However, miss rate can be a misleading measure for several reasons. Therefore some designers prefer measuring *misses per instruction* rather than misses per memory reference (miss rate). These two are related:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

(This equation is often expressed in integers rather than fractions, as misses per 1000 instructions.)

The problem with both measures is that they don't factor in the cost of a miss. A better measure is the *average memory access time*,

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where *hit time* is the time to hit in the cache and *miss penalty* is the time to replace the block from memory (that is, the cost of a miss). Average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time. In Chapter 3 we will see that speculative processors may execute other instructions during a miss, thereby reducing the effective miss penalty. The use of multithreading (introduced in Chapter 3) also allows a processor to tolerate misses without being forced to idle. As we will examine shortly, to take advantage of such latency tolerating techniques, we need caches that can service requests while handling an outstanding miss.

If this material is new to you, or if this quick review moves too quickly, see Appendix B. It covers the same introductory material in more depth and includes examples of caches from real computers and quantitative evaluations of their effectiveness.

Section B.3 in Appendix B presents six basic cache optimizations, which we quickly review here. The appendix also gives quantitative examples of the benefits of these optimizations. We also comment briefly on the power implications of these trade-offs.

1. *Larger block size to reduce miss rate*—The simplest way to reduce the miss rate is to take advantage of spatial locality and increase the block size. Larger blocks

reduce compulsory misses, but they also increase the miss penalty. Because larger blocks lower the number of tags, they can slightly reduce static power. Larger block sizes can also increase capacity or conflict misses, especially in smaller caches. Choosing the right block size is a complex trade-off that depends on the size of cache and the miss penalty.

2. *Bigger caches to reduce miss rate*—The obvious way to reduce capacity misses is to increase cache capacity. Drawbacks include potentially longer hit time of the larger cache memory and higher cost and power. Larger caches increase both static and dynamic power.
3. *Higher associativity to reduce miss rate*—Obviously, increasing associativity reduces conflict misses. Greater associativity can come at the cost of increased hit time. As we will see shortly, associativity also increases power consumption.
4. *Multilevel caches to reduce miss penalty*—A difficult decision is whether to make the cache hit time fast, to keep pace with the high clock rate of processors, or to make the cache large to reduce the gap between the processor accesses and main memory accesses. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match a fast clock cycle time, yet the second-level (or third-level) cache can be large enough to capture many accesses that would go to main memory. The focus on misses in second-level caches leads to larger blocks, bigger capacity, and higher associativity. Multilevel caches are more power-efficient than a single aggregate cache. If L1 and L2 refer, respectively, to first- and second-level caches, we can redefine the average memory access time:

$$\text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

5. *Giving priority to read misses over writes to reduce miss penalty*—A write buffer is a good place to implement this optimization. Write buffers create hazards because they hold the updated value of a location needed on a read miss—that is, a read-after-write hazard through memory. One solution is to check the contents of the write buffer on a read miss. If there are no conflicts, and if the memory system is available, sending the read before the writes reduces the miss penalty. Most processors give reads priority over writes. This choice has little effect on power consumption.
6. *Avoiding address translation during indexing of the cache to reduce hit time*—Caches must cope with the translation of a virtual address from the processor to a physical address to access memory. (Virtual memory is covered in Sections 2.4 and B.4.) A common optimization is to use the page offset—the part that is identical in both virtual and physical addresses—to index the cache, as described in Appendix B, page B.38. This virtual index/physical tag method introduces some system complications and/or limitations on the size and structure of the L1 cache, but the advantages of removing the translation lookaside buffer (TLB) access from the critical path outweigh the disadvantages.

Note that each of the preceding six optimizations has a potential disadvantage that can lead to increased, rather than decreased, average memory access time.

The rest of this chapter assumes familiarity with the preceding material and the details in Appendix B. In the "Putting It All Together" section, we examine the memory hierarchy for a microprocessor designed for a high-end desktop or smaller server, the Intel Core i7 6700, as well as one designed for use in a PMD, the Arm Cortex-53, which is the basis for the processor used in several tablets and smartphones. Within each of these classes, there is a significant diversity in approach because of the intended use of the computer.

Although the i7 6700 has more cores and bigger caches than the Intel processors designed for mobile uses, the processors have similar architectures. A processor designed for small servers, such as the i7 6700, or larger servers, such as the Intel Xeon processors, typically is running a large number of concurrent processes, often for different users. Thus memory bandwidth becomes more important, and these processors offer larger caches and more aggressive memory systems to boost that bandwidth.

In contrast, PMDs not only serve one user but generally also have smaller operating systems, usually less multitasking (running of several applications simultaneously), and simpler applications. PMDs must consider both performance and energy consumption, which determines battery life. Before we dive into more advanced cache organizations and optimizations, one needs to understand the various memory technologies and how they are evolving.

2.2 Memory Technology and Optimizations

> …the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. …Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large. (p. 209)
>
> **Maurice Wilkes.**
> *Memoirs of a Computer Pioneer* (1985)

This section describes the technologies used in a memory hierarchy, specifically in building caches and main memory. These technologies are SRAM (static random-access memory), DRAM (dynamic random-access memory), and Flash. The last of these is used as an alternative to hard disks, but because its characteristics are based on semiconductor technology, it is appropriate to include in this section.

Using SRAM addresses the need to minimize access time to caches. When a cache miss occurs, however, we need to move the data from the main memory as quickly as possible, which requires a high bandwidth memory. This high memory bandwidth can be achieved by organizing the many DRAM chips that make up the main memory into multiple memory banks and by making the memory bus wider, or by doing both.

To allow memory systems to keep up with the bandwidth demands of modern processors, memory innovations started happening inside the DRAM chips themselves. This section describes the technology inside the memory chips and those innovative, internal organizations. Before describing the technologies and options, we need to introduce some terminology.

With the introduction of burst transfer memories, now widely used in both Flash and DRAM, memory latency is quoted using two measures—access time and cycle time. *Access time* is the time between when a read is requested and when the desired word arrives, and *cycle time* is the minimum time between unrelated requests to memory.

Virtually all computers since 1975 have used DRAMs for main memory and SRAMs for cache, with one to three levels integrated onto the processor chip with the CPU. PMDs must balance power and performance, and because they have more modest storage needs, PMDs use Flash rather than disk drives, a decision increasingly being followed by desktop computers as well.

SRAM Technology

The first letter of SRAM stands for *static*. The dynamic nature of the circuits in DRAM requires data to be written back after being read—thus the difference between the access time and the cycle time as well as the need to refresh. SRAMs don't need to refresh, so the access time is very close to the cycle time. SRAMs typically use six transistors per bit to prevent the information from being disturbed when read. SRAM needs only minimal power to retain the charge in standby mode.

In earlier times, most desktop and server systems used SRAM chips for their primary, secondary, or tertiary caches. Today, all three levels of caches are integrated onto the processor chip. In high-end server chips, there may be as many as 24 cores and up to 60 MiB of cache; such systems are often configured with 128–256 GiB of DRAM per processor chip. The access times for large, third-level, on-chip caches are typically two to eight times that of a second-level cache. Even so, the L3 access time is usually at least five times faster than a DRAM access.

On-chip, cache SRAMs are normally organized with a width that matches the block size of the cache, with the tags stored in parallel to each block. This allows an entire block to be read out or written into a single cycle. This capability is particularly useful when writing data fetched after a miss into the cache or when writing back a block that must be evicted from the cache. The access time to the cache (ignoring the hit detection and selection in a set associative cache) is proportional to the number of blocks in the cache, whereas the energy consumption depends both on the number of bits in the cache (static power) and on the number of blocks (dynamic power). Set associative caches reduce the initial access time to the memory because the size of the memory is smaller, but increase the time for hit detection and block selection, a topic we will cover in Section 2.3.

DRAM Technology

As early DRAMs grew in capacity, the cost of a package with all the necessary address lines was an issue. The solution was to multiplex the address lines, thereby

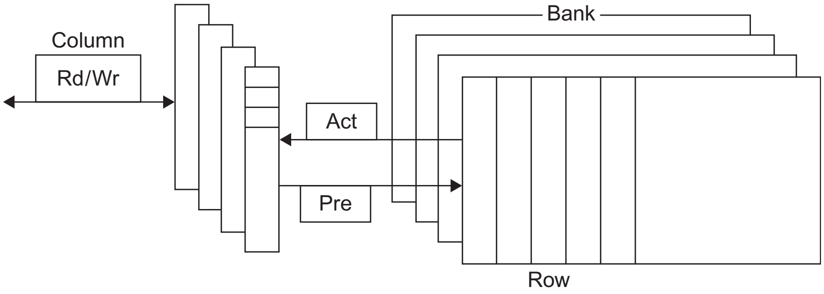


Figure 2.3 Internal organization of a DRAM. Modern DRAMs are organized in banks, up to 16 for DDR4. Each bank consists of a series of rows. Sending an ACT (Activate) command opens a bank and a row and loads the row into a row buffer. When the row is in the buffer, it can be transferred by successive column addresses at whatever the width of the DRAM is (typically 4, 8, or 16 bits in DDR4) or by specifying a block transfer and the starting address. The Precharge commend (PRE) closes the bank and row and readies it for a new access. Each command, as well as block transfers, are synchronized with a clock. See the next section discussing SDRAM. The row and column signals are sometimes called RAS and CAS, based on the original names of the signals.

cutting the number of address pins in half. Figure 2.3 shows the basic DRAM organization. One-half of the address is sent first during the *row access strobe* (RAS). The other half of the address, sent during the *column access strobe* (CAS), follows it. These names come from the internal chip organization, because the memory is organized as a rectangular matrix addressed by rows and columns.

An additional requirement of DRAM derives from the property signified by its first letter, *D*, for *dynamic*. To pack more bits per chip, DRAMs use only a single transistor, which effectively acts as a capacitor, to store a bit. This has two implications: first, the sensing wires that detect the charge must be precharged, which sets them "halfway" between a logical 0 and 1, allowing the small charge stored in the cell to cause a 0 or 1 to be detected by the sense amplifiers. On reading, a row is placed into a row buffer, where CAS signals can select a portion of the row to read out from the DRAM. Because reading a row destroys the information, it must be written back when the row is no longer needed. This write back happens in overlapped fashion, but in early DRAMs, it meant that the cycle time before a new row could be read was larger than the time to read a row and access a portion of that row.

In addition, to prevent loss of information as the charge in a cell leaks away (assuming it is not read or written), each bit must be "refreshed" periodically. Fortunately, all the bits in a row can be refreshed simultaneously just by reading that row and writing it back. Therefore every DRAM in the memory system must access every row within a certain time window, such as 64 ms. DRAM controllers include hardware to refresh the DRAMs periodically.

This requirement means that the memory system is occasionally unavailable because it is sending a signal telling every chip to refresh. The time for a refresh is a row activation and a precharge that also writes the row back (which takes roughly 2/3 of the time to get a datum because no column select is needed), and this is required for each row of the DRAM. Because the memory matrix in a DRAM is conceptually square, the number of steps in a refresh is usually the square root of the DRAM capacity. DRAM designers try to keep time spent refreshing to less than 5% of the total time. So far we have presented main memory as if it operated like a Swiss train, consistently delivering the goods exactly according to schedule. In fact, with SDRAMs, a DRAM controller (usually on the processor chip) tries to optimize accesses by avoiding opening new rows and using block transfer when possible. Refresh adds another unpredictable factor.

Amdahl suggested as a rule of thumb that memory capacity should grow linearly with processor speed to keep a balanced system. Thus a 1000 MIPS processor should have 1000 MiB of memory. Processor designers rely on DRAMs to supply that demand. In the past, they expected a fourfold improvement in capacity every three years, or 55% per year. Unfortunately, the performance of DRAMs is growing at a much slower rate. The slower performance improvements arise primarily because of smaller decreases in the row access time, which is determined by issues such as power limitations and the charge capacity (and thus the size) of an individual memory cell. Before we discuss these performance trends in more detail, we need to describe the major changes that occurred in DRAMs starting in the mid-1990s.

Improving Memory Performance Inside a DRAM Chip: SDRAMs

Although very early DRAMs included a buffer allowing multiple column accesses to a single row, without requiring a new row access, they used an asynchronous interface, which meant that every column access and transfer involved overhead to synchronize with the controller. In the mid-1990s, designers added a clock signal to the DRAM interface so that the repeated transfers would not bear that overhead, thereby creating *synchronous DRAM* (SDRAM). In addition to reducing overhead, SDRAMs allowed the addition of a burst transfer mode where multiple transfers can occur without specifying a new column address. Typically, eight or more 16-bit transfers can occur without sending any new addresses by placing the DRAM in burst mode. The inclusion of such burst mode transfers has meant that there is a significant gap between the bandwidth for a stream of random accesses versus access to a block of data.

To overcome the problem of getting more bandwidth from the memory as DRAM density increased, DRAMS were made wider. Initially, they offered a four-bit transfer mode; in 2017, DDR2, DDR3, and DDR DRAMS had up to 4, 8, or 16 bit buses.

In the early 2000s, a further innovation was introduced: *double data rate* (DDR), which allows a DRAM to transfer data both on the rising and the falling edge of the memory clock, thereby doubling the peak data rate.

Finally, SDRAMs introduced *banks* to help with power management, improve access time, and allow interleaved and overlapped accesses to different banks.

Access to different banks can be overlapped with each other, and each bank has its own row buffer. Creating multiple banks inside a DRAM effectively adds another segment to the address, which now consists of bank number, row address, and column address. When an address is sent that designates a new bank, that bank must be opened, incurring an additional delay. The management of banks and row buffers is completely handled by modern memory control interfaces, so that when a subsequent access specifies the same row for an open bank, the access can happen quickly, sending only the column address.

To initiate a new access, the DRAM controller sends a bank and row number (called *Activate* in SDRAMs and formerly called RAS—row select). That command opens the row and reads the entire row into a buffer. A column address can then be sent, and the SDRAM can transfer one or more data items, depending on whether it is a single item request or a burst request. Before accessing a new row, the bank must be precharged. If the row is in the same bank, then the precharge delay is seen; however, if the row is in another bank, closing the row and precharging can overlap with accessing the new row. In synchronous DRAMs, each of these command cycles requires an integral number of clock cycles.

From 1980 to 1995, DRAMs scaled with Moore's Law, doubling capacity every 18 months (or a factor of 4 in 3 years). From the mid-1990s to 2010, capacity increased more slowly with roughly 26 months between a doubling. From 2010 to 2016, capacity only doubled! Figure 2.4 shows the capacity and access time for various generations of DDR SDRAMs. From DDR1 to DDR3, access times improved by a factor of about 3, or about 7% per year. DDR4 improves power and bandwidth over DDR3, but has similar access latency.

As Figure 2.4 shows, DDR is a sequence of standards. DDR2 lowers power from DDR1 by dropping the voltage from 2.5 to 1.8 V and offers higher clock rates: 266, 333, and 400 MHz. DDR3 drops voltage to 1.5 V and has a maximum clock speed of 800 MHz. (As we discuss in the next section, GDDR5 is a graphics

			Best case access time (no precharge)			Precharge needed
Production year	**Chip size**	**DRAM type**	**RAS time (ns)**	**CAS time (ns)**	**Total (ns)**	**Total (ns)**
2000	256M bit	DDR1	21	21	42	63
2002	512M bit	DDR1	15	15	30	45
2004	1G bit	DDR2	15	15	30	45
2006	2G bit	DDR2	10	10	20	30
2010	4G bit	DDR3	13	13	26	39
2016	8G bit	DDR4	13	13	26	39

Figure 2.4 Capacity and access times for DDR SDRAMs by year of production. Access time is for a random memory word and assumes a new row must be opened. If the row is in a different bank, we assume the bank is precharged; if the row is not open, then a precharge is required, and the access time is longer. As the number of banks has increased, the ability to hide the precharge time has also increased. DDR4 SDRAMs were initially expected in 2014, but did not begin production until early 2016.

Standard	I/O clock rate	M transfers/s	DRAM name	MiB/s/DIMM	DIMM name
DDR1	133	266	DDR266	2128	PC2100
DDR1	150	300	DDR300	2400	PC2400
DDR1	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10,664	PC10700
DDR3	800	1600	DDR3-1600	12,800	PC12800
DDR4	1333	2666	DDR4-2666	21,300	PC21300

Figure 2.5 Clock rates, bandwidth, and names of DDR DRAMS and DIMMs in 2016. Note the numerical relationship between the columns. The third column is twice the second, and the fourth uses the number from the third column in the name of the DRAM chip. The fifth column is eight times the third column, and a rounded version of this number is used in the name of the DIMM. DDR4 saw significant first use in 2016.

RAM and is based on DDR3 DRAMs.) DDR4, which shipped in volume in early 2016, but was expected in 2014, drops the voltage to 1–1.2 V and has a maximum expected clock rate of 1600 MHz. DDR5 is unlikely to reach production quantities until 2020 or later.

With the introduction of DDR, memory designers increasing focused on bandwidth, because improvements in access time were difficult. Wider DRAMs, burst transfers, and double data rate all contributed to rapid increases in memory bandwidth. DRAMs are commonly sold on small boards called *dual inline memory modules* (DIMMs) that contain 4–16 DRAM chips and that are normally organized to be 8 bytes wide (+ ECC) for desktop and server systems. When DDR SDRAMs are packaged as DIMMs, they are confusingly labeled by the peak *DIMM* bandwidth. Therefore the DIMM name PC3200 comes from 200 MHz × 2 × 8 bytes, or 3200 MiB/s; it is populated with DDR SDRAM chips. Sustaining the confusion, the chips themselves are labeled with *the number of bits per second* rather than their clock rate, so a 200 MHz DDR chip is called a DDR400. Figure 2.5 shows the relationships' I/O clock rate, transfers per second per chip, chip bandwidth, chip name, DIMM bandwidth, and DIMM name.

Reducing Power Consumption in SDRAMs

Power consumption in dynamic memory chips consists of both dynamic power used in a read or write and static or standby power; both depend on the operating voltage. In the most advanced DDR4 SDRAMs, the operating voltage has dropped to 1.2 V, significantly reducing power versus DDR2 and DDR3 SDRAMs. The addition of banks also reduced power because only the row in a single bank is read.

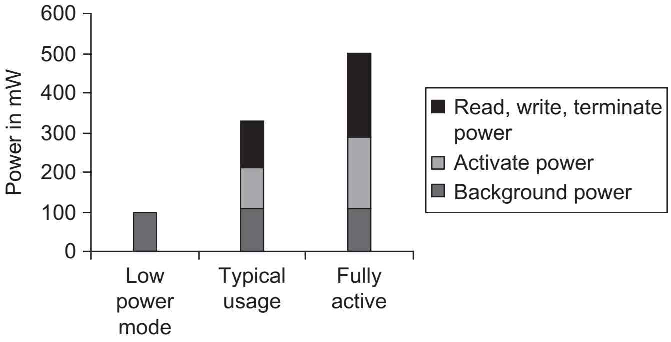


Figure 2.6 Power consumption for a DDR3 SDRAM operating under three conditions: low-power (shutdown) mode, typical system mode (DRAM is active 30% of the time for reads and 15% for writes), and fully active mode, where the DRAM is continuously reading or writing. Reads and writes assume bursts of eight transfers. These data are based on a Micron 1.5V 2GB DDR3-1066, although similar savings occur in DDR4 SDRAMs.

In addition to these changes, all recent SDRAMs support a power-down mode, which is entered by telling the DRAM to ignore the clock. Power-down mode disables the SDRAM, except for internal automatic refresh (without which entering power-down mode for longer than the refresh time will cause the contents of memory to be lost). Figure 2.6 shows the power consumption for three situations in a 2 GB DDR3 SDRAM. The exact delay required to return from low power mode depends on the SDRAM, but a typical delay is 200 SDRAM clock cycles.

Graphics Data RAMs

GDRAMs or GSDRAMs (Graphics or Graphics Synchronous DRAMs) are a special class of DRAMs based on SDRAM designs but tailored for handling the higher bandwidth demands of graphics processing units. GDDR5 is based on DDR3 with earlier GDDRs based on DDR2. Because graphics processor units (GPUs; see Chapter 4) require more bandwidth per DRAM chip than CPUs, GDDRs have several important differences:

1. GDDRs have wider interfaces: 32-bits versus 4, 8, or 16 in current designs.
2. GDDRs have a higher maximum clock rate on the data pins. To allow a higher transfer rate without incurring signaling problems, GDRAMS normally connect directly to the GPU and are attached by soldering them to the board, unlike DRAMs, which are normally arranged in an expandable array of DIMMs.

Altogether, these characteristics let GDDRs run at two to five times the bandwidth per DRAM versus DDR3 DRAMs.

Packaging Innovation: Stacked or Embedded DRAMs

The newest innovation in 2017 in DRAMs is a packaging innovation, rather than a circuit innovation. It places multiple DRAMs in a stacked or adjacent fashion embedded within the same package as the processor. (Embedded DRAM also is used to refer to designs that place DRAM on the processor chip.) Placing the DRAM and processor in the same package lowers access latency (by shortening the delay between the DRAMs and the processor) and potentially increases bandwidth by allowing more and faster connections between the processor and DRAM; thus several producers have called it *high bandwidth memory (HBM)*.

One version of this technology places the DRAM die directly on the CPU die using solder bump technology to connect them. Assuming adequate heat management, multiple DRAM dies can be stacked in this fashion. Another approach stacks only DRAMs and abuts them with the CPU in a single package using a substrate (interposer) containing the connections. Figure 2.7 shows these two different interconnection schemes. Prototypes of HBM that allow stacking of up to eight chips have been demonstrated. With special versions of SDRAMs, such a package could contain 8 GiB of memory and have data transfer rates of 1 TB/s. The 2.5D technique is currently available. Because the chips must be specifically manufactured to stack, it is quite likely that most early uses will be in high-end server chipsets.

In some applications, it may be possible to internally package enough DRAM to satisfy the needs of the application. For example, a version of an Nvidia GPU used as a node in a special-purpose cluster design is being developed using HBM, and it is likely that HBM will become a successor to GDDR5 for higher-end applications. In some cases, it may be possible to use HBM as main memory, although the cost limitations and heat removal issues currently rule out this technology for some embedded applications. In the next section, we consider the possibility of using HBM as an additional level of cache.

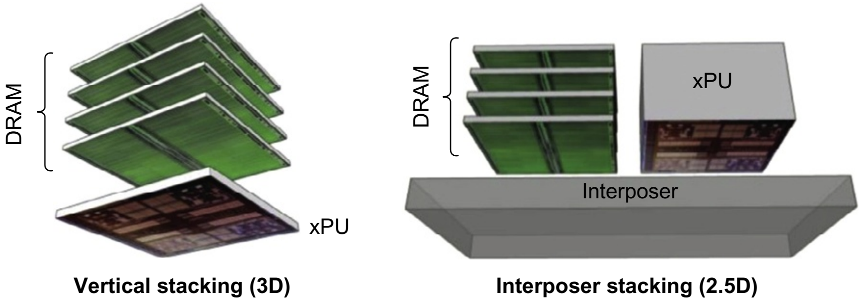


Figure 2.7 Two forms of die stacking. The 2.5D form is available now. 3D stacking is under development and faces heat management challenges due to the CPU.

Flash Memory

Flash memory is a type of EEPROM (electronically erasable programmable read-only memory), which is normally read-only but can be erased. The other key property of Flash memory is that it holds its contents without any power. We focus on NAND Flash, which has higher density than NOR Flash and is more suitable for large-scale nonvolatile memories; the drawback is that access is sequential and writing is slower, as we explain below.

Flash is used as the secondary storage in PMDs in the same manner that a disk functions in a laptop or server. In addition, because most PMDs have a limited amount of DRAM, Flash may also act as a level of the memory hierarchy, to a much greater extent than it might have to do in a desktop or server with a main memory that might be 10–100 times larger.

Flash uses a very different architecture and has different properties than standard DRAM. The most important differences are

1. Reads to Flash are sequential and read an entire page, which can be 512 bytes, 2 KiB, or 4 KiB. Thus NAND Flash has a long delay to access the first byte from a random address (about 25 μS), but can supply the remainder of a page block at about 40 MiB/s. By comparison, a DDR4 SDRAM takes about 40 ns to the first byte and can transfer the rest of the row at 4.8 GiB/s. Comparing the time to transfer 2 KiB, NAND Flash takes about 75 μS, while DDR SDRAM takes less than 500 ns, making Flash about 150 times slower. Compared to magnetic disk, however, a 2 KiB read from Flash is 300 to 500 times faster. From these numbers, we can see why Flash is not a candidate to replace DRAM for main memory, but is a candidate to replace magnetic disk.
2. Flash memory must be erased (thus the name flash for the "flash" erase process) before it is overwritten, and it is erased in blocks rather than individual bytes or words. This requirement means that when data must be written to Flash, an entire block must be assembled, either as new data or by merging the data to be written and the rest of the block's contents. For writing, Flash is about 1500 times slower then SDRAM, and about 8–15 times as fast as magnetic disk.
3. Flash memory is nonvolatile (i.e., it keeps its contents even when power is not applied) and draws significantly less power when not reading or writing (from less than half in standby mode to zero when completely inactive).
4. Flash memory limits the number of times that any given block can be written, typically at least 100,000. By ensuring uniform distribution of written blocks throughout the memory, a system can maximize the lifetime of a Flash memory system. This technique, called *write leveling*, is handled by Flash memory controllers.
5. High-density NAND Flash is cheaper than SDRAM but more expensive than disks: roughly $2/GiB for Flash, $20 to $40/GiB for SDRAM, and $0.09/GiB for magnetic disks. In the past five years, Flash has decreased in cost at a rate that is almost twice as fast as that of magnetic disks.

Like DRAM, Flash chips include redundant blocks to allow chips with small numbers of defects to be used; the remapping of blocks is handled in the Flash chip. Flash controllers handle page transfers, provide caching of pages, and handle write leveling.

The rapid improvements in high-density Flash have been critical to the development of low-power PMDs and laptops, but they have also significantly changed both desktops, which increasingly use solid state disks, and large servers, which often combine disk and Flash-based storage.

Phase-Change Memory Technology

Phase-change memory (PCM) has been an active research area for decades. The technology typically uses a small heating element to change the state of a bulk substrate between its crystalline form and an amorphous form, which have different resistive properties. Each bit corresponds to a crosspoint in a two-dimensional network that overlays the substrate. Reading is done by sensing the resistance between an x and y point (thus the alternative name *memristor*), and writing is accomplished by applying a current to change the phase of the material. The absence of an active device (such as a transistor) should lead to lower costs and greater density than that of NAND Flash.

In 2017 Micron and Intel began delivering Xpoint memory chips that are believed to be based on PCM. The technology is expected to have much better write durability than NAND Flash and, by eliminating the need to erase a page before writing, achieve an increase in write performance versus NAND of up to a factor of ten. Read latency is also better than Flash by perhaps a factor of 2–3. Initially, it is expected to be priced slightly higher than Flash, but the advantages in write performance and write durability may make it attractive, especially for SSDs. Should this technology scale well and be able to achieve additional cost reductions, it may be the solid state technology that will depose magnetic disks, which have reigned as the primary bulk nonvolatile store for more than 50 years.

Enhancing Dependability in Memory Systems

Large caches and main memories significantly increase the possibility of errors occurring both during the fabrication process and dynamically during operation. Errors that arise from a change in circuitry and are repeatable are called *hard errors* or *permanent faults.* Hard errors can occur during fabrication, as well as from a circuit change during operation (e.g., failure of a Flash memory cell after many writes). All DRAMs, Flash memory, and most SRAMs are manufactured with spare rows so that a small number of manufacturing defects can be accommodated by programming the replacement of a defective row by a spare row. Dynamic errors, which are changes to a cell's contents, not a change in the circuitry, are called *soft errors* or *transient faults*.

Dynamic errors can be detected by parity bits and detected and fixed by the use of error correcting codes (ECCs). Because instruction caches are read-only, parity suffices. In larger data caches and in main memory, ECC is used to allow errors to be both detected and corrected. Parity requires only one bit of overhead to detect a single error in a sequence of bits. Because a multibit error would be undetected with parity, the number of bits protected by a parity bit must be limited. One parity bit per 8 data bits is a typical ratio. ECC can detect two errors and correct a single error with a cost of 8 bits of overhead per 64 data bits.

In very large systems, the possibility of multiple errors as well as complete failure of a single memory chip becomes significant. Chipkill was introduced by IBM to solve this problem, and many very large systems, such as IBM and SUN servers and the Google Clusters, use this technology. (Intel calls their version SDDC.) Similar in nature to the RAID approach used for disks, Chipkill distributes the data and ECC information so that the complete failure of a single memory chip can be handled by supporting the reconstruction of the missing data from the remaining memory chips. Using an analysis by IBM and assuming a 10,000 processor server with 4 GiB per processor yields the following rates of unrecoverable errors in three years of operation:

- Parity only: About 90,000, or one unrecoverable (or undetected) failure every 17 minutes.
- ECC only: About 3500, or about one undetected or unrecoverable failure every 7.5 hours.
- Chipkill: About one undetected or unrecoverable failure every 2 months.

Another way to look at this is to find the maximum number of servers (each with 4 GiB) that can be protected while achieving the same error rate as demonstrated for Chipkill. For parity, even a server with only one processor will have an unrecoverable error rate higher than a 10,000-server Chipkill protected system. For ECC, a 17-server system would have about the same failure rate as a 10,000-server Chipkill system. Therefore Chipkill is a requirement for the 50,000–100,00 servers in warehouse-scale computers (see Section 6.8 of Chapter 6).

2.3 Ten Advanced Optimizations of Cache Performance

The preceding average memory access time formula gives us three metrics for cache optimizations: hit time, miss rate, and miss penalty. Given the recent trends, we add cache bandwidth and power consumption to this list. We can classify the 10 advanced cache optimizations we examine into five categories based on these metrics:

1. *Reducing the hit time*—Small and simple first-level caches and way-prediction. Both techniques also generally decrease power consumption.
2. *Increasing cache bandwidth*—Pipelined caches, multibanked caches, and nonblocking caches. These techniques have varying impacts on power consumption.

3. *Reducing the miss penalty*—Critical word first and merging write buffers. These optimizations have little impact on power.
4. *Reducing the miss rate*—Compiler optimizations. Obviously any improvement at compile time improves power consumption.
5. *Reducing the miss penalty or miss rate via parallelism*—Hardware prefetching and compiler prefetching. These optimizations generally increase power consumption, primarily because of prefetched data that are unused.

In general, the hardware complexity increases as we go through these optimizations. In addition, several of the optimizations require sophisticated compiler technology, and the final one depends on HBM. We will conclude with a summary of the implementation complexity and the performance benefits of the 10 techniques presented in Figure 2.18 on page 113. Because some of these are straightforward, we cover them briefly; others require more description.

First Optimization: Small and Simple First-Level Caches to Reduce Hit Time and Power

The pressure of both a fast clock cycle and power limitations encourages limited size for first-level caches. Similarly, use of lower levels of associativity can reduce both hit time and power, although such trade-offs are more complex than those involving size.

The critical timing path in a cache hit is the three-step process of addressing the tag memory using the index portion of the address, comparing the read tag value to the address, and setting the multiplexor to choose the correct data item if the cache is set associative. Direct-mapped caches can overlap the tag check with the transmission of the data, effectively reducing hit time. Furthermore, lower levels of associativity will usually reduce power because fewer cache lines must be accessed.

Although the total amount of on-chip cache has increased dramatically with new generations of microprocessors, because of the clock rate impact arising from a larger L1 cache, the size of the L1 caches has recently increased either slightly or not at all. In many recent processors, designers have opted for more associativity rather than larger caches. An additional consideration in choosing the associativity is the possibility of eliminating address aliases; we discuss this topic shortly.

One approach to determining the impact on hit time and power consumption in advance of building a chip is to use CAD tools. CACTI is a program to estimate the access time and energy consumption of alternative cache structures on CMOS microprocessors within 10% of more detailed CAD tools. For a given minimum feature size, CACTI estimates the hit time of caches as a function of cache size, associativity, number of read/write ports, and more complex parameters. Figure 2.8 shows the estimated impact on hit time as cache size and associativity are varied. Depending on cache size, for these parameters, the model suggests that the hit time for direct mapped is slightly faster than two-way set associative and that two-way set associative is 1.2 times as fast as four-way and four-way is 1.4

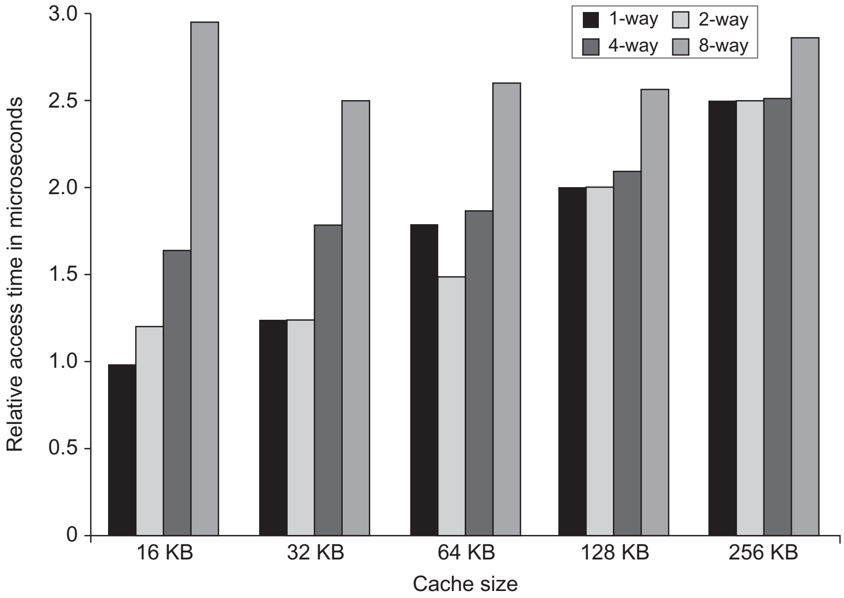


Figure 2.8 Relative access times generally increase as cache size and associativity are increased. These data come from the CACTI model 6.5 by Tarjan et al. (2005). The data assume typical embedded SRAM technology, a single bank, and 64-byte blocks. The assumptions about cache layout and the complex trade-offs between interconnect delays (that depend on the size of a cache block being accessed) and the cost of tag checks and multiplexing lead to results that are occasionally surprising, such as the lower access time for a 64 KiB with two-way set associativity versus direct mapping. Similarly, the results with eight-way set associativity generate unusual behavior as cache size is increased. Because such observations are highly dependent on technology and detailed design assumptions, tools such as CACTI serve to reduce the search space. These results are relative; nonetheless, they are likely to shift as we move to more recent and denser semiconductor technologies.

times as fast as eight-way. Of course, these estimates depend on technology as well as the size of the cache, and CACTI must be carefully aligned with the technology; Figure 2.8 shows the relative tradeoffs for one technology.

Example Using the data in Figure B.8 in Appendix B and Figure 2.8, determine whether a 32 KiB four-way set associative L1 cache has a faster memory access time than a 32 KiB two-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

Answer Let the access time for the two-way set associative cache be 1. Then, for the two-way cache,

$$\begin{aligned}\text{Average memory access time}_{\text{2-way}} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}\\ &= 1 + 0.038 \times 15 = 1.38\end{aligned}$$

For the four-way cache, the access time is 1.4 times longer. The elapsed time of the miss penalty is 15/1.4 = 10.1. Assume 10 for simplicity:

$$\begin{aligned}\text{Average memory access time}_{\text{4-way}} &= \text{Hit time}_{\text{2-way}} \times 1.4 + \text{Miss rate} \times \text{Miss penalty}\\ &= 1.4 + 0.037 \times 10 = 1.77\end{aligned}$$

Clearly, the higher associativity looks like a bad trade-off; however, because cache access in modern processors is often pipelined, the exact impact on the clock cycle time is difficult to assess.

Energy consumption is also a consideration in choosing both the cache size and associativity, as Figure 2.9 shows. The energy cost of higher associativity ranges from more than a factor of 2 to negligible in caches of 128 or 256 KiB when going from direct mapped to two-way set associative.

As energy consumption has become critical, designers have focused on ways to reduce the energy needed for cache access. In addition to associativity, the other key factor in determining the energy used in a cache access is the number of blocks in the cache because it determines the number of "rows" that are accessed. A designer could reduce the number of rows by increasing the block size (holding total cache size constant), but this could increase the miss rate, especially in smaller L1 caches.

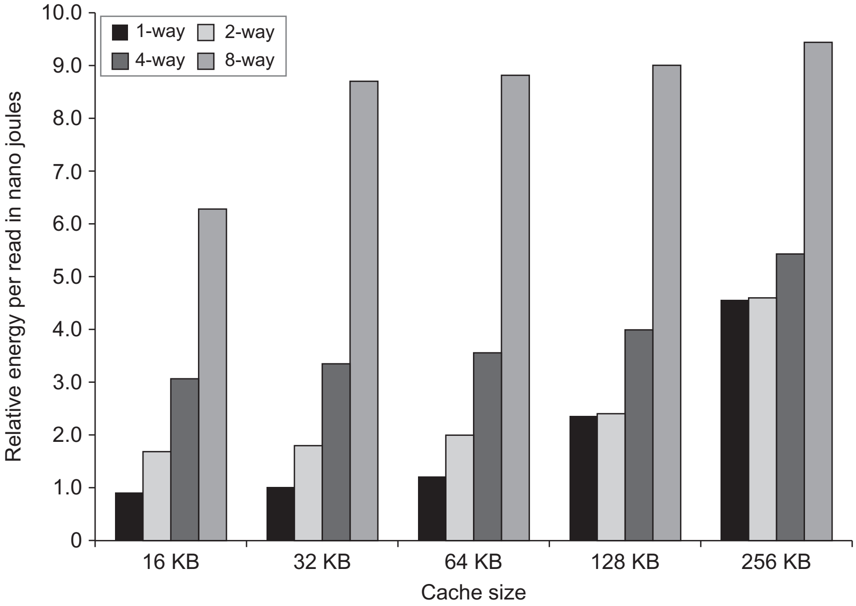


Figure 2.9 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.

An alternative is to organize the cache in banks so that an access activates only a portion of the cache, namely the bank where the desired block resides. The primary use of multibanked caches is to increase the bandwidth of the cache, an optimization we consider shortly. Multibanking also reduces energy because less of the cache is accessed. The L3 caches in many multicores are logically unified, but physically distributed, and effectively act as a multibanked cache. Based on the address of a request, only one of the physical L3 caches (a bank) is actually accessed. We discuss this organization further in Chapter 5.

In recent designs, there are three other factors that have led to the use of higher associativity in first-level caches despite the energy and access time costs. First, many processors take at least 2 clock cycles to access the cache and thus the impact of a longer hit time may not be critical. Second, to keep the TLB out of the critical path (a delay that would be larger than that associated with increased associativity), almost all L1 caches should be virtually indexed. This limits the size of the cache to the page size times the associativity because then only the bits within the page are used for the index. There are other solutions to the problem of indexing the cache before address translation is completed, but increasing the associativity, which also has other benefits, is the most attractive. Third, with the introduction of multithreading (see Chapter 3), conflict misses can increase, making higher associativity more attractive.

Second Optimization: Way Prediction to Reduce Hit Time

Another approach reduces conflict misses and yet maintains the hit speed of direct-mapped cache. In *way prediction*, extra bits are kept in the cache to predict the way (or block within the set) of the *next* cache access. This prediction means the multiplexor is set early to select the desired block, and in that clock cycle, only a single tag comparison is performed in parallel with reading the cache data. A miss results in checking the other blocks for matches in the next clock cycle.

Added to each block of a cache are block predictor bits. The bits select which of the blocks to try on the next cache access. If the predictor is correct, the cache access latency is the fast hit time. If not, it tries the other block, changes the way predictor, and has a latency of one extra clock cycle. Simulations suggest that set prediction accuracy is in excess of 90% for a two-way set associative cache and 80% for a four-way set associative cache, with better accuracy on I-caches than D-caches. Way prediction yields lower average memory access time for a two-way set associative cache if it is at least 10% faster, which is quite likely. Way prediction was first used in the MIPS R10000 in the mid-1990s. It is popular in processors that use two-way set associativity and was used in several ARM processors, which have four-way set associative caches. For very fast processors, it may be challenging to implement the one-cycle stall that is critical to keeping the way prediction penalty small.

An extended form of way prediction can also be used to reduce power consumption by using the way prediction bits to decide which cache block to actually access (the way prediction bits are essentially extra address bits); this approach, which might be called way selection, saves power when the way prediction is correct but adds significant time on a way misprediction, because the access, not just the tag match and selection, must be repeated. Such an optimization is likely to make sense only in low-power processors. Inoue et al. (1999) estimated that using the way selection approach with a four-way set associative cache increases the average access time for the I-cache by 1.04 and for the D-cache by 1.13 on the SPEC95 benchmarks, but it yields an average cache power consumption relative to a normal four-way set associative cache that is 0.28 for the I-cache and 0.35 for the D-cache. One significant drawback for way selection is that it makes it difficult to pipeline the cache access; however, as energy concerns have mounted, schemes that do not require powering up the entire cache make increasing sense.

Example Assume that there are half as many D-cache accesses as I-cache accesses and that the I-cache and D-cache are responsible for 25% and 15% of the processor's power consumption in a normal four-way set associative implementation. Determine if way selection improves performance per watt based on the estimates from the preceding study.

Answer For the I-cache, the savings in power is $25 \times 0.28 = 0.07$ of the total power, while for the D-cache it is $15 \times 0.35 = 0.05$ for a total savings of 0.12. The way prediction version requires 0.88 of the power requirement of the standard four-way cache. The increase in cache access time is the increase in I-cache average access time plus one-half the increase in D-cache access time, or $1.04 + 0.5 \times 0.13 = 1.11$ times longer. This result means that way selection has 0.90 of the performance of a standard four-way cache. Thus way selection improves performance per joule very slightly by a ratio of $0.90/0.88 = 1.02$. This optimization is best used where power rather than performance is the key objective.

Third Optimization: Pipelined Access and Multibanked Caches to Increase Bandwidth

These optimizations increase cache bandwidth either by pipelining the cache access or by widening the cache with multiple banks to allow multiple accesses per clock; these optimizations are the dual to the superpipelined and superscalar approaches to increasing instruction throughput. These optimizations are primarily targeted at L1, where access bandwidth constrains instruction throughput. Multiple banks are also used in L2 and L3 caches, but primarily as a power-management technique.

Pipelining L1 allows a higher clock cycle, at the cost of increased latency. For example, the pipeline for the instruction cache access for Intel Pentium processors in the mid-1990s took 1 clock cycle; for the Pentium Pro through Pentium III in the mid-1990s through 2000, it took 2 clock cycles; and for the Pentium 4, which became available in 2000, and the current Intel Core i7, it takes 4 clock cycles. Pipelining the instruction cache effectively increases the number of pipeline stages,

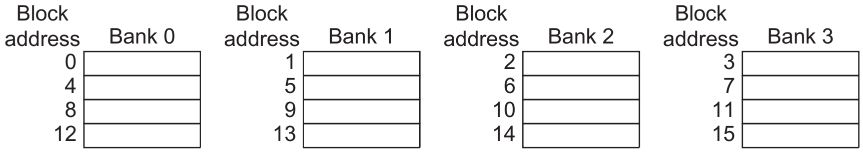


Figure 2.10 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per block, each of these addresses would be multiplied by 64 to get byte addressing.

leading to a greater penalty on mispredicted branches. Correspondingly, pipelining the data cache leads to more clock cycles between issuing the load and using the data (see Chapter 3). Today, all processors use some pipelining of L1, if only for the simple case of separating the access and hit detection, and many high-speed processors have three or more levels of cache pipelining.

It is easier to pipeline the instruction cache than the data cache because the processor can rely on high performance branch prediction to limit the latency effects. Many superscalar processors can issue and execute more than one memory reference per clock (allowing a load or store is common, and some processors allow multiple loads). To handle multiple data cache accesses per clock, we can divide the cache into independent banks, each supporting an independent access. Banks were originally used to improve performance of main memory and are now used inside modern DRAM chips as well as with caches. The Intel Core i7 has four banks in L1 (to support up to 2 memory accesses per clock).

Clearly, banking works best when the accesses naturally spread themselves across the banks, so the mapping of addresses to banks affects the behavior of the memory system. A simple mapping that works well is to spread the addresses of the block sequentially across the banks, which is called *sequential interleaving*. For example, if there are four banks, bank 0 has all blocks whose address modulo 4 is 0, bank 1 has all blocks whose address modulo 4 is 1, and so on. Figure 2.10 shows this interleaving. Multiple banks also are a way to reduce power consumption in both caches and DRAM.

Multiple banks are also useful in L2 or L3 caches, but for a different reason. With multiple banks in L2, we can handle more than one outstanding L1 miss, if the banks do not conflict. This is a key capability to support nonblocking caches, our next optimization. The L2 in the Intel Core i7 has eight banks, while Arm Cortex processors have used L2 caches with 1–4 banks. As mentioned earlier, multibanking can also reduce energy consumption.

Fourth Optimization: Nonblocking Caches to Increase Cache Bandwidth

For pipelined computers that allow out-of-order execution (discussed in Chapter 3), the processor need not stall on a data cache miss. For example, the processor could continue fetching instructions from the instruction cache while waiting for the data cache to return the missing data. A *nonblocking cache* or *lockup-free cache* escalates the potential benefits of such a scheme by allowing the data cache to continue to supply cache hits during a miss. This "hit under miss" optimization reduces the effective miss penalty by being helpful during a miss instead of ignoring the requests of the processor. A subtle and complex option is that the cache may further lower the effective miss penalty if it can overlap multiple misses: a "hit under multiple miss" or "miss under miss" optimization. The second option is beneficial only if the memory system can service multiple misses; most high-performance processors (such as the Intel Core processors) usually support both, whereas many lower-end processors provide only limited nonblocking support in L2.

To examine the effectiveness of nonblocking caches in reducing the cache miss penalty, Farkas and Jouppi (1994) did a study assuming 8 KiB caches with a 14-cycle miss penalty (appropriate for the early 1990s). They observed a reduction in the effective miss penalty of 20% for the SPECINT92 benchmarks and 30% for the SPECFP92 benchmarks when allowing one hit under miss.

Li et al. (2011) updated this study to use a multilevel cache, more modern assumptions about miss penalties, and the larger and more demanding SPECCPU2006 benchmarks. The study was done assuming a model based on a single core of an Intel i7 (see Section 2.6) running the SPECCPU2006 benchmarks. Figure 2.11 shows the reduction in data cache access latency when allowing 1, 2, and 64 hits under a miss; the caption describes further details of the memory system. The larger caches and the addition of an L3 cache since the earlier study have reduced the benefits with the SPECINT2006 benchmarks showing an average reduction in cache latency of about 9% and the SPECFP2006 benchmarks about 12.5%.

Example Which is more important for floating-point programs: two-way set associativity or hit under one miss for the primary data caches? What about integer programs? Assume the following average miss rates for 32 KiB data caches: 5.2% for floating-point programs with a direct-mapped cache, 4.9% for the programs with a two-way set associative cache, 3.5% for integer programs with a direct-mapped cache, and 3.2% for integer programs with a two-way set associative cache. Assume the miss penalty to L2 is 10 cycles, and the L2 misses and penalties are the same.

Answer For floating-point programs, the average memory stall times are

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 5.2\% \times 10 = 0.52$$

$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 4.9\% \times 10 = 0.49$$

The cache access latency (including stalls) for two-way associativity is 0.49/0.52 or 94% of direct-mapped cache. Figure 2.11 caption indicates that a hit under one miss reduces the average data cache access latency for floating-point programs to 87.5% of a blocking cache. Therefore, for floating-point programs, the

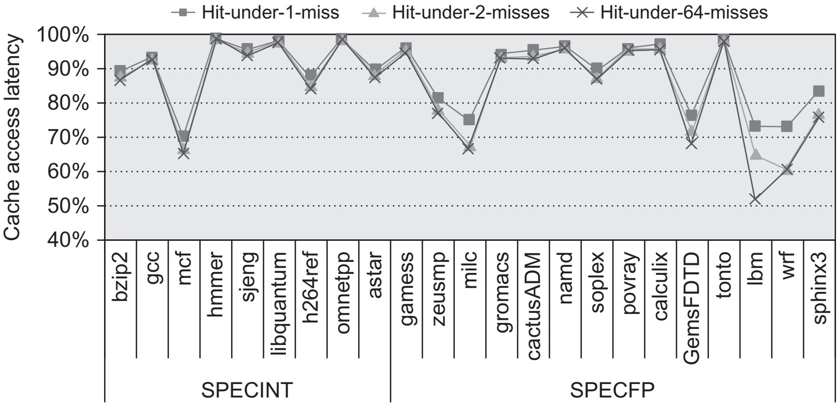


Figure 2.11 The effectiveness of a nonblocking cache is evaluated by allowing 1, 2, or 64 hits under a cache miss with 9 SPECINT (on the left) and 9 SPECFP (on the right) benchmarks. The data memory system modeled after the Intel i7 consists of a 32 KiB L1 cache with a four-cycle access latency. The L2 cache (shared with instructions) is 256 KiB with a 10-clock cycle access latency. The L3 is 2 MiB and a 36-cycle access latency. All the caches are eight-way set associative and have a 64-byte block size. Allowing one hit under miss reduces the miss penalty by 9% for the integer benchmarks and 12.5% for the floating point. Allowing a second hit improves these results to 10% and 16%, and allowing 64 results in little additional improvement.

direct-mapped data cache supporting one hit under one miss gives better performance than a two-way set-associative cache that blocks on a miss.

For integer programs, the calculation is

$$\text{Miss rate}_{\text{DM}} \times \text{Miss penalty} = 3.5\% \times 10 = 0.35$$
$$\text{Miss rate}_{\text{2-way}} \times \text{Miss penalty} = 3.2\% \times 10 = 0.32$$

The data cache access latency of a two-way set associative cache is thus 0.32/0.35 or 91% of direct-mapped cache, while the reduction in access latency when allowing a hit under one miss is 9%, making the two choices about equal.

The real difficulty with performance evaluation of nonblocking caches is that a cache miss does not necessarily stall the processor. In this case, it is difficult to judge the impact of any single miss and thus to calculate the average memory access time. The effective miss penalty is not the sum of the misses but the nonoverlapped time that the processor is stalled. The benefit of nonblocking caches is complex, as it depends upon the miss penalty when there are multiple misses, the memory reference pattern, and how many instructions the processor can execute with a miss outstanding.

In general, out-of-order processors are capable of hiding much of the miss penalty of an L1 data cache miss that hits in the L2 cache but are not capable of hiding a significant fraction of a lower-level cache miss. Deciding how many outstanding misses to support depends on a variety of factors:

- The temporal and spatial locality in the miss stream, which determines whether a miss can initiate a new access to a lower-level cache or to memory.
- The bandwidth of the responding memory or cache.
- To allow more outstanding misses at the lowest level of the cache (where the miss time is the longest) requires supporting at least that many misses at a higher level, because the miss must initiate at the highest level cache.
- The latency of the memory system.

The following simplified example illustrates the key idea.

Example Assume a main memory access time of 36 ns and a memory system capable of a sustained transfer rate of 16 GiB/s. If the block size is 64 bytes, what is the maximum number of outstanding misses we need to support assuming that we can maintain the peak bandwidth given the request stream and that accesses never conflict. If the probability of a reference colliding with one of the previous four is 50%, and we assume that the access has to wait until the earlier access completes, estimate the number of maximum outstanding references. For simplicity, ignore the time between misses.

Answer In the first case, assuming that we can maintain the peak bandwidth, the memory system can support $(16 \times 10)^9/64 = 250$ million references per second. Because each reference takes 36 ns, we can support $250 \times 10^6 \times 36 \times 10^{-9} = 9$ references. If the probability of a collision is greater than 0, then we need more outstanding references, because we cannot start work on those colliding references; the memory system needs more independent references, not fewer! To approximate, we can simply assume that half the memory references do not have to be issued to the memory. This means that we must support twice as many outstanding references, or 18.

In Li, Chen, Brockman, and Jouppi's study, they found that the reduction in CPI for the integer programs was about 7% for one hit under miss and about 12.7% for 64. For the floating-point programs, the reductions were 12.7% for one hit under miss and 17.8% for 64. These reductions track fairly closely the reductions in the data cache access latency shown in Figure 2.11.

Implementing a Nonblocking Cache

Although nonblocking caches have the potential to improve performance, they are nontrivial to implement. Two initial types of challenges arise: arbitrating contention between hits and misses, and tracking outstanding misses so that we know when loads or stores can proceed. Consider the first problem. In a blocking cache, misses cause the processor to stall and no further accesses to the cache will occur until the miss is handled. In a nonblocking cache, however, hits can collide with misses returning from the next level of the memory hierarchy. If we allow multiple outstanding misses, which almost all recent processors do, it is even possible for misses to collide. These collisions must be resolved, usually by first giving priority to hits over misses, and second by ordering colliding misses (if they can occur).

The second problem arises because we need to track multiple outstanding misses. In a blocking cache, we always know which miss is returning, because only one can be outstanding. In a nonblocking cache, this is rarely true. At first glance, you might think that misses always return in order, so that a simple queue could be kept to match a returning miss with the longest outstanding request. Consider, however, a miss that occurs in L1. It may generate either a hit or miss in L2; if L2 is also nonblocking, then the order in which misses are returned to L1 will not necessarily be the same as the order in which they originally occurred. Multicore and other multiprocessor systems that have nonuniform cache access times also introduce this complication.

When a miss returns, the processor must know which load or store caused the miss, so that instruction can now go forward; and it must know where in the cache the data should be placed (as well as the setting of tags for that block). In recent processors, this information is kept in a set of registers, typically called the *Miss Status Handling Registers (MSHRs)*. If we allow *n* outstanding misses, there will be *n* MSHRs, each holding the information about where a miss goes in the cache and the value of any tag bits for that miss, as well as the information indicating which load or store caused the miss (in the next chapter, you will see how this is tracked). Thus, when a miss occurs, we allocate an MSHR for handling that miss, enter the appropriate information about the miss, and tag the memory request with the index of the MSHR. The memory system uses that tag when it returns the data, allowing the cache system to transfer the data and tag information to the appropriate cache block and "notify" the load or store that generated the miss that the data is now available and that it can resume operation. Nonblocking caches clearly require extra logic and thus have some cost in energy. It is difficult, however, to assess their energy costs exactly because they may reduce stall time, thereby decreasing execution time and resulting energy consumption.

In addition to the preceding issues, multiprocessor memory systems, whether within a single chip or on multiple chips, must also deal with complex implementation issues related to memory coherency and consistency. Also, because cache misses are no longer atomic (because the request and response are split and may be interleaved among multiple requests), there are possibilities for deadlock. For the interested reader, Section I.7 in online Appendix I deals with these issues in detail.

Fifth Optimization: Critical Word First and Early Restart to Reduce Miss Penalty

This technique is based on the observation that the processor normally needs just one word of the block at a time. This strategy is impatience: don't wait for the full block to be loaded before sending the requested word and restarting the processor. Here are two specific strategies:

- *Critical word first*—Request the missed word first from memory and send it to the processor as soon as it arrives; let the processor continue execution while filling the rest of the words in the block.
- *Early restart*—Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the processor and let the processor continue execution.

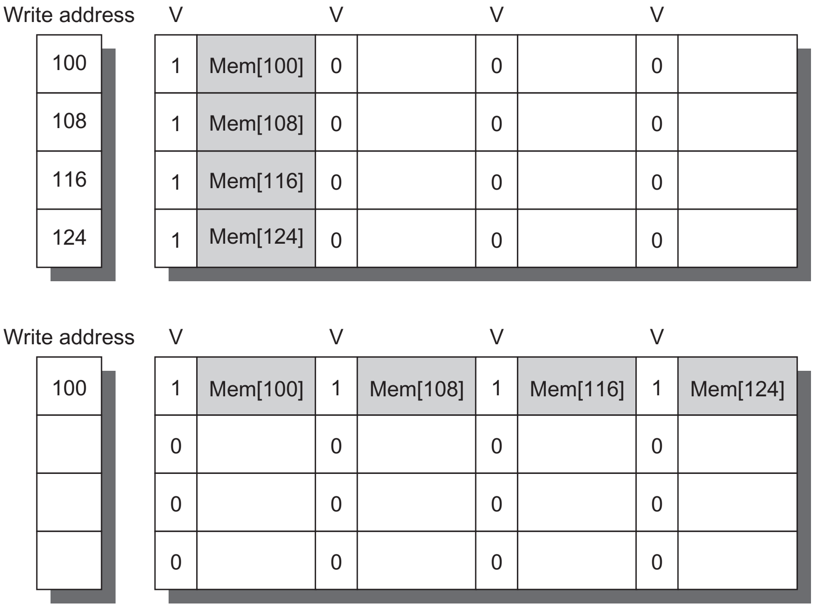
Generally, these techniques only benefit designs with large cache blocks because the benefit is low unless blocks are large. Note that caches normally continue to satisfy accesses to other blocks while the rest of the block is being filled.

However, given spatial locality, there is a good chance that the next reference is to the rest of the block. Just as with nonblocking caches, the miss penalty is not simple to calculate. When there is a second request in critical word first, the effective miss penalty is the nonoverlapped time from the reference until the second piece arrives. The benefits of critical word first and early restart depend on the size of the block and the likelihood of another access to the portion of the block that has not yet been fetched. For example, for SPECint2006 running on the i7 6700, which uses early restart and critical word first, there is more than one reference made to a block with an outstanding miss (1.23 references on average with a range from 0.5 to 3.0). We explore the performance of the i7 memory hierarchy in more detail in Section 2.6.

Sixth Optimization: Merging Write Buffer to Reduce Miss Penalty

Write-through caches rely on write buffers, as all stores must be sent to the next lower level of the hierarchy. Even write-back caches use a simple buffer when a block is replaced. If the write buffer is empty, the data and the full address are written in the buffer, and the write is finished from the processor's perspective; the processor continues working while the write buffer prepares to write the word to memory. If the buffer contains other modified blocks, the addresses can be checked to see if the address of the new data matches the address of a valid write buffer entry. If so, the new data are combined with that entry. *Write merging* is the name of this optimization. The Intel Core i7, among many others, uses write merging.

If the buffer is full and there is no address match, the cache (and processor) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently because multiword writes are usually faster than writes performed one word at a time. Skadron and Clark (1997) found that even a merging four-entry write buffer generated stalls that led to a 5%–10% performance loss.



Write address	V		V		V		V	
100	1	Mem[100]	0		0		0	
108	1	Mem[108]	0		0		0	
116	1	Mem[116]	0		0		0	
124	1	Mem[124]	0		0		0	

Write address	V		V		V		V	
100	1	Mem[100]	1	Mem[108]	1	Mem[116]	1	Mem[124]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Figure 2.12 In this illustration of write merging, the write buffer on top does not use write merging while the write buffer on the bottom does. The four writes are merged into a single buffer entry with write merging; without it, the buffer is full even though three-fourths of each entry is wasted. The buffer has four entries, and each entry holds four 64-bit words. The address for each entry is on the left, with a valid bit (V) indicating whether the next sequential 8 bytes in this entry are occupied. (Without write merging, the words to the right in the upper part of the figure would be used only for instructions that wrote multiple words at the same time.)

The optimization also reduces stalls because of the write buffer being full. Figure 2.12 shows a write buffer with and without write merging. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged fit exactly within a single entry of the write buffer.

Note that input/output device registers are often mapped into the physical address space. These I/O addresses *cannot* allow write merging because separate I/O registers may not act like an array of words in memory. For example, they may require one address and data word per I/O register rather than use multiword writes using a single address. These side effects are typically implemented by marking the pages as requiring nonmerging write through by the caches.

Seventh Optimization: Compiler Optimizations to Reduce Miss Rate

Thus far, our techniques have required changing the hardware. This next technique reduces miss rates without any hardware changes.

This magical reduction comes from optimized software—the hardware designer's favorite solution! The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again, research is split between improvements in instruction misses and improvements in data misses. The optimizations presented next are found in many modern compilers.

Loop Interchange

Some programs have nested loops that access data in memory in nonsequential order. Simply exchanging the nesting of the loops can make the code access the data in the order in which they are stored. Assuming the arrays do not fit in the cache, this technique reduces misses by improving spatial locality; reordering maximizes use of data in a cache block before they are discarded. For example, if `x` is a two-dimensional array of size [5000,100] allocated so that `x[i,j]` and `x[i,j+1]` are adjacent (an order called row major because the array is laid out by rows), then the two pieces of the following code show how the accesses can be optimized:

```
/* Before */
for (j = 0; j < 100; j = j + 1)
        for (i = 0; i < 5000; i = i + 1)
                x[i][j] = 2 * x[i][j];
/* After */
for (i = 0; i < 5000; i = i + 1)
        for (j = 0; j < 100; j = j + 1)
                x[i][j] = 2 * x[i][j];
```

The original code would skip through memory in strides of 100 words, while the revised version accesses all the words in one cache block before going to the next block. This optimization improves cache performance without affecting the number of instructions executed.

Blocking

This optimization improves temporal locality to reduce misses. We are again dealing with multiple arrays, with some arrays accessed by rows and some by columns. Storing the arrays row by row (*row major order*) or column by column (*column major order*) does not solve the problem because both rows and columns are used in every loop iteration. Such orthogonal accesses mean that transformations such as loop interchange still leave plenty of room for improvement.

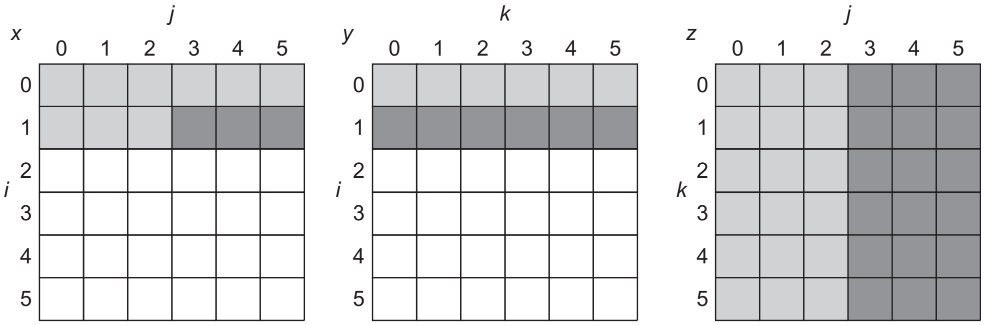


Figure 2.13 A snapshot of the three arrays x, y, and z when $N = 6$ and i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of y and z are read repeatedly to calculate new elements of x. The variables i, j, and k are shown along the rows or columns used to access the arrays.

Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*. The goal is to maximize accesses to the data loaded into the cache before the data are replaced. The following code example, which performs matrix multiplication, helps motivate the optimization:

```
/* Before */
for (i = 0; i < N; i = i + 1)
      for (j = 0; j < N; j = j + 1)
           {r = 0;
            for (k = 0; k < N; k = k + 1)
                 r = r + y[i][k]*z[k][j];
            x[i][j] = r;
           };
```

The two inner loops read all N-by-N elements of z, read the same N elements in a row of y repeatedly, and write one row of N elements of x. Figure 2.13 gives a snapshot of the accesses to the three arrays. A dark shade indicates a recent access, a light shade indicates an older access, and white means not yet accessed.

The number of capacity misses clearly depends on N and the size of the cache. If it can hold all three N-by-N matrices, then all is well, provided there are no cache conflicts. If the cache can hold one N-by-N matrix and one row of N, then at least the ith row of y and the array z may stay in the cache. Less than that and misses may occur for both x and z. In the worst case, there would be $2N^3 + N^2$ memory words accessed for N^3 operations.

To ensure that the elements being accessed can fit in the cache, the original code is changed to compute on a submatrix of size B by B. Two inner loops now compute in steps of size B rather than the full length of x and z. B is called the *blocking factor*. (Assume x is initialized to zero.)

```
/* After */
for (jj = 0; jj < N; jj = jj + B)
for (kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i + 1)
      for (j = jj; j < min(jj + B,N); j = j + 1)
          {r = 0;
           for (k = kk; k < min(kk + B,N); k = k + 1)
                r = r + y[i][k]*z[k][j];
           x[i][j] = x[i][j] + r;
          };
```

Figure 2.14 illustrates the accesses to the three arrays using blocking. Looking only at capacity misses, the total number of memory words accessed is $2N^3/B + N^2$. This total is an improvement by an approximate factor of B. Therefore blocking exploits a combination of spatial and temporal locality, because y benefits from spatial locality and z benefits from temporal locality. Although our example uses a square block (BxB), we could also use a rectangular block, which would be necessary if the matrix were not square.

Although we have aimed at reducing cache misses, blocking can also be used to help register allocation. By taking a small blocking size such that the block can be held in registers, we can minimize the number of loads and stores in the program.

As we shall see in Section 4.8 of Chapter 4, cache blocking is absolutely necessary to get good performance from cache-based processors running applications using matrices as the primary data structure.

Eighth Optimization: Hardware Prefetching of Instructions and Data to Reduce Miss Penalty or Miss Rate

Nonblocking caches effectively reduce the miss penalty by overlapping execution with memory access. Another approach is to prefetch items before the processor requests them. Both instructions and data can be prefetched, either directly into

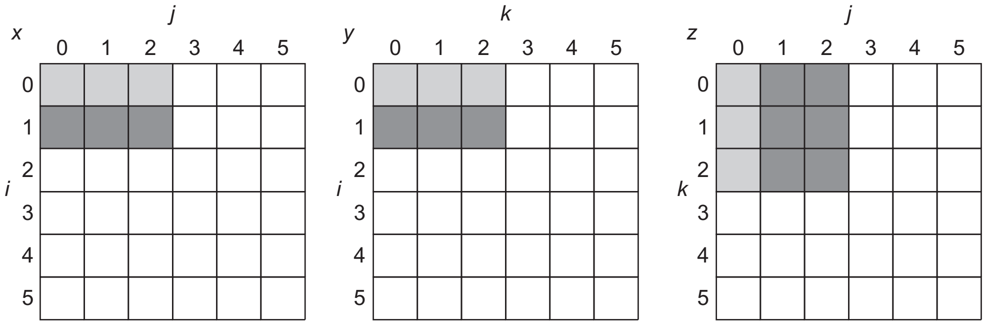


Figure 2.14 The age of accesses to the arrays x, y, and z when *B* = 3. Note that, in contrast to Figure 2.13, a smaller number of elements is accessed.

the caches or into an external buffer that can be more quickly accessed than main memory.

Instruction prefetch is frequently done in hardware outside of the cache. Typically, the processor fetches two blocks on a miss: the requested block and the next consecutive block. The requested block is placed in the instruction cache when it returns, and the prefetched block is placed in the instruction stream buffer. If the requested block is present in the instruction stream buffer, the original cache request is canceled, the block is read from the stream buffer, and the next prefetch request is issued.

A similar approach can be applied to data accesses (Jouppi, 1990). Palacharla and Kessler (1994) looked at a set of scientific programs and considered multiple stream buffers that could handle either instructions or data. They found that eight stream buffers could capture 50%–70% of all misses from a processor with two 64 KiB four-way set associative caches, one for instructions and the other for data.

The Intel Core i7 supports hardware prefetching into both L1 and L2 with the most common case of prefetching being accessing the next line. Some earlier Intel processors used more aggressive hardware prefetching, but that resulted in reduced performance for some applications, causing some sophisticated users to turn off the capability.

Figure 2.15 shows the overall performance improvement for a subset of SPEC2000 programs when hardware prefetching is turned on. Note that this figure

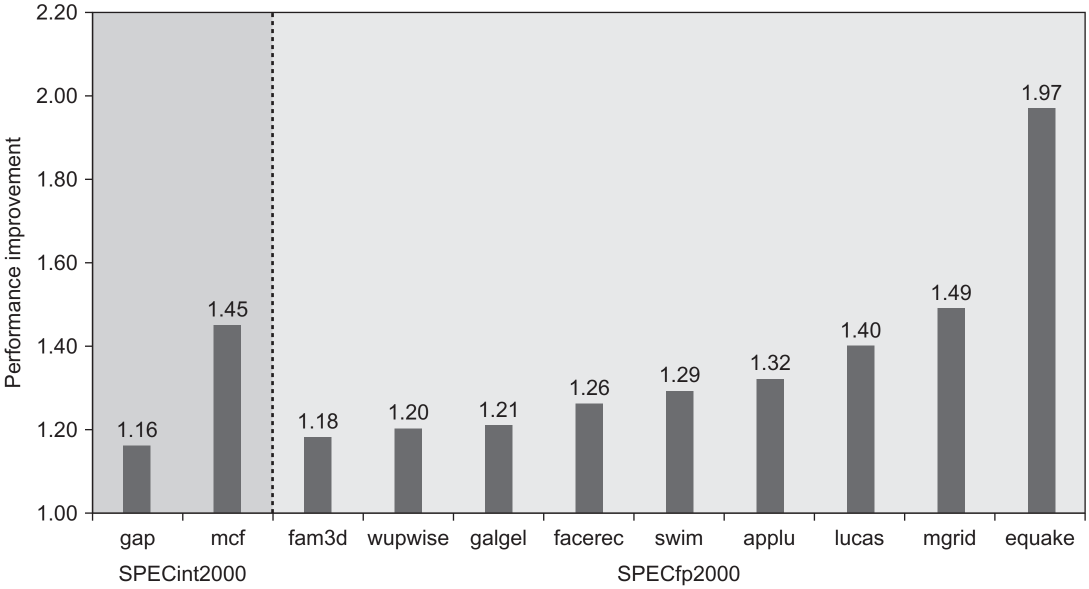


Figure 2.15 Speedup because of hardware prefetching on Intel Pentium 4 with hardware prefetching turned on for 2 of 12 SPECint2000 benchmarks and 9 of 14 SPECfp2000 benchmarks. Only the programs that benefit the most from prefetching are shown; prefetching speeds up the missing 15 SPECCPU benchmarks by less than 15% (Boggs et al., 2004).

includes only 2 of 12 integer programs, while it includes the majority of the SPECCPU floating-point programs. We will return to our evaluation of prefetching on the i7 in Section 2.6.

Prefetching relies on utilizing memory bandwidth that otherwise would be unused, but if it interferes with demand misses, it can actually lower performance. Help from compilers can reduce useless prefetching. When prefetching works well, its impact on power is negligible. When prefetched data are not used or useful data are displaced, prefetching will have a very negative impact on power.

Ninth Optimization: Compiler-Controlled Prefetching to Reduce Miss Penalty or Miss Rate

An alternative to hardware prefetching is for the compiler to insert prefetch instructions to request data before the processor needs it. There are two flavors of prefetch:

- *Register prefetch* loads the value into a register.
- *Cache prefetch* loads data only into the cache and not the register.

Either of these can be *faulting* or *nonfaulting*; that is, the address does or does not cause an exception for virtual address faults and protection violations. Using this terminology, a normal load instruction could be considered a "faulting register prefetch instruction." Nonfaulting prefetches simply turn into no-ops if they would normally result in an exception, which is what we want.

The most effective prefetch is "semantically invisible" to a program: it doesn't change the contents of registers and memory, *and* it cannot cause virtual memory faults. Most processors today offer nonfaulting cache prefetches. This section assumes nonfaulting cache prefetch, also called *nonbinding* prefetch.

Prefetching makes sense only if the processor can proceed while prefetching the data; that is, the caches do not stall but continue to supply instructions and data while waiting for the prefetched data to return. As you would expect, the data cache for such computers is normally nonblocking.

Like hardware-controlled prefetching, the goal is to overlap execution with the prefetching of data. Loops are the important targets because they lend themselves to prefetch optimizations. If the miss penalty is small, the compiler just unrolls the loop once or twice, and it schedules the prefetches with the execution. If the miss penalty is large, it uses software pipelining (see Appendix H) or unrolls many times to prefetch data for a future iteration.

Issuing prefetch instructions incurs an instruction overhead, however, so compilers must take care to ensure that such overheads do not exceed the benefits. By concentrating on references that are likely to be cache misses, programs can avoid unnecessary prefetches while improving average memory access time significantly.

Example For the following code, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching. Let's assume we have an 8 KiB direct-mapped data cache with 16-byte blocks, and it is a write-back cache that does write allocate. The elements of `a` and `b` are 8 bytes long because they are double-precision floating-point arrays. There are 3 rows and 100 columns for `a` and 101 rows and 3 columns for `b`. Let's also assume they are not in the cache at the start of the program.

```
for (i = 0; i < 3; i = i + 1)
        for (j = 0; j < 100; j = j + 1)
                a[i][j] = b[j][0] * b[j + 1][0];
```

Answer The compiler will first determine which accesses are likely to cause cache misses; otherwise, we will waste time on issuing prefetch instructions for data that would be hits. Elements of `a` are written in the order that they are stored in memory, so `a` will benefit from spatial locality: The even values of `j` will miss and the odd values will hit. Because `a` has 3 rows and 100 columns, its accesses will lead to $3 \times (100/2)$, or 150 misses.

The array `b` does not benefit from spatial locality because the accesses are not in the order it is stored. The array `b` does benefit twice from temporal locality: the same elements are accessed for each iteration of `i`, and each iteration of `j` uses the same value of `b` as the last iteration. Ignoring potential conflict misses, the misses because of `b` will be for `b[j+1][0]` accesses when `i` = 0, and also the first access to `b[j][0]` when `j` = 0. Because `j` goes from 0 to 99 when `i` = 0, accesses to `b` lead to 100 + 1, or 101 misses.

Thus this loop will miss the data cache approximately 150 times for `a` plus 101 times for `b`, or 251 misses.

To simplify our optimization, we will not worry about prefetching the first accesses of the loop. These may already be in the cache, or we will pay the miss penalty of the first few elements of `a` or `b`. Nor will we worry about suppressing the prefetches at the end of the loop that try to prefetch beyond the end of `a` (`a[i][100]` … `a[i][106]`) and the end of `b` (`b[101][0]` … `b[107][0]`). If these were faulting prefetches, we could not take this luxury. Let's assume that the miss penalty is so large we need to start prefetching at least, say, seven iterations in advance. (Stated alternatively, we assume prefetching has no benefit until the eighth iteration.) We underline the changes to the preceding code needed to add prefetching.

```
for (j = 0; j < 100; j = j + 1) {
        prefetch(b[j + 7][0]);
        /* b(j,0) for 7 iterations later */
        prefetch(a[0][j + 7]);
        /* a(0,j) for 7 iterations later */
        a[0][j] = b[j][0] * b[j + 1][0];};
```

```
for (i = 1; i < 3; i = i + 1)
      for (j = 0; j < 100; j = j + 1) {
            prefetch(a[i][j + 7]);
            /* a(i,j) for + 7 iterations */
            a[i][j] = b[j][0] * b[j + 1][0];}
```

This revised code prefetches `a[i][7]` through `a[i][99]` and `b[7][0]` through `b[100][0]`, reducing the number of nonprefetched misses to

- 7 misses for elements `b[0][0]`, `b[1][0]`, … , `b[6][0]` in the first loop
- 4 misses (⌈7/2⌉) for elements `a[0][0]`, `a[0][1]`, … , `a[0][6]` in the first loop (spatial locality reduces misses to 1 per 16-byte cache block)
- 4 misses (⌈7/2⌉) for elements `a[1][0]`, `a[1][1]`, … , `a[1][6]` in the second loop
- 4 misses (⌈7/2⌉) for elements `a[2][0]`, `a[2][1]`, … , `a[2][6]` in the second loop

or a total of 19 nonprefetched misses. The cost of avoiding 232 cache misses is executing 400 prefetch instructions, likely a good trade-off.

Example Calculate the time saved in the preceding example. Ignore instruction cache misses and assume there are no conflict or capacity misses in the data cache. Assume that prefetches can overlap with each other and with cache misses, thereby transferring at the maximum memory bandwidth. Here are the key loop times ignoring cache misses: the original loop takes 7 clock cycles per iteration, the first prefetch loop takes 9 clock cycles per iteration, and the second prefetch loop takes 8 clock cycles per iteration (including the overhead of the outer for loop). A miss takes 100 clock cycles.

Answer The original doubly nested loop executes the multiply 3×100 or 300 times. Because the loop takes 7 clock cycles per iteration, the total is 300×7 or 2100 clock cycles plus cache misses. Cache misses add 251×100 or 25,100 clock cycles, giving a total of 27,200 clock cycles. The first prefetch loop iterates 100 times; at 9 clock cycles per iteration the total is 900 clock cycles plus cache misses. Now add 11×100 or 1100 clock cycles for cache misses, giving a total of 2000. The second loop executes 2×100 or 200 times, and at 8 clock cycles per iteration, it takes 1600 clock cycles plus 8×100 or 800 clock cycles for cache misses. This gives a total of 2400 clock cycles. From the prior example, we know that this code executes 400 prefetch instructions during the $2000 + 2400$ or 4400 clock cycles to execute these two loops. If we assume that the prefetches are completely overlapped with the rest of the execution, then the prefetch code is 27,200/4400, or 6.2 times faster.

Although array optimizations are easy to understand, modern programs are more likely to use pointers. Luk and Mowry (1999) have demonstrated that compiler-based prefetching can sometimes be extended to pointers as well. Of 10 programs with recursive data structures, prefetching all pointers when a node is visited improved performance by 4%–31% in half of the programs. On the other hand, the remaining programs were still within 2% of their original performance. The issue is both whether prefetches are to data already in the cache and whether they occur early enough for the data to arrive by the time it is needed.

Many processors support instructions for cache prefetch, and high-end processors (such as the Intel Core i7) often also do some type of automated prefetch in hardware.

Tenth Optimization: Using HBM to Extend the Memory Hierarchy

Because most general-purpose processors in servers will likely want more memory than can be packaged with HBM packaging, it has been proposed that the in-package DRAMs be used to build massive L4 caches, with upcoming technologies ranging from 128 MiB to 1 GiB and more, considerably more than current on-chip L3 caches. Using such large DRAM-based caches raises an issue: where do the tags reside? That depends on the number of tags. Suppose we were to use a 64B block size; then a 1 GiB L4 cache requires 96 MiB of tags—far more static memory than exists in the caches on the CPU. Increasing the block size to 4 KiB, yields a dramatically reduced tag store of 256 K entries or less than 1 MiB total storage, which is probably acceptable, given L3 caches of 4–16 MiB or more in next-generation, multicore processors. Such large block sizes, however, have two major problems.

First, the cache may be used inefficiently when content of many blocks are not needed; this is called the *fragmentation problem*, and it also occurs in virtual memory systems. Furthermore, transferring such large blocks is inefficient if much of the data is unused. Second, because of the large block size, the number of distinct blocks held in the DRAM cache is much lower, which can result in more misses, especially for conflict and consistency misses.

One partial solution to the first problem is to add *sublocking*. *Subblocking* allow parts of the block to be invalid, requiring that they be fetched on a miss. Subblocking, however, does nothing to address the second problem.

The tag storage is the major drawback for using a smaller block size. One possible solution for that difficulty is to store the tags for L4 in the HBM. At first glance this seems unworkable, because it requires two accesses to DRAM for each L4 access: one for the tags and one for the data itself. Because of the long access time for random DRAM accesses, typically 100 or more processor clock cycles, such an approach had been discarded. Loh and Hill (2011) proposed a clever solution to this problem: place the tags and the data in the same row in the HBM SDRAM. Although opening the row (and eventually closing it) takes a large amount of time, the CAS latency to access a different part of the row is about one-third the new row access time. Thus we can access the tag portion of the block first, and if it is a hit, then use a column access to choose the correct word. Loh and Hill (L-H) have proposed organizing the L4 HBM cache so that each SDRAM row consists of a set of tags (at the head of the block) and 29 data segments, making a 29-way set associative cache. When L4 is accessed, the appropriate row is opened and the tags are read; a hit requires one more column access to get the matching data.

Qureshi and Loh (2012) proposed an improvement called an *alloy cache* that reduces the hit time. An *alloy cache* molds the tag and data together and uses a direct mapped cache structure. This allows the L4 access time to be reduced to a single HBM cycle by directly indexing the HBM cache and doing a burst transfer of both the tag and data. Figure 2.16 shows the hit latency for the alloy cache, the L-H scheme, and SRAM based tags. The alloy cache reduces hit time by more than a factor of 2 versus the L-H scheme, in return for an increase in the miss rate by a factor of 1.1–1.2. The choice of benchmarks is explained in the caption.

Unfortunately, in both schemes, misses require two full DRAM accesses: one to get the initial tag and a follow-on access to the main memory (which is even

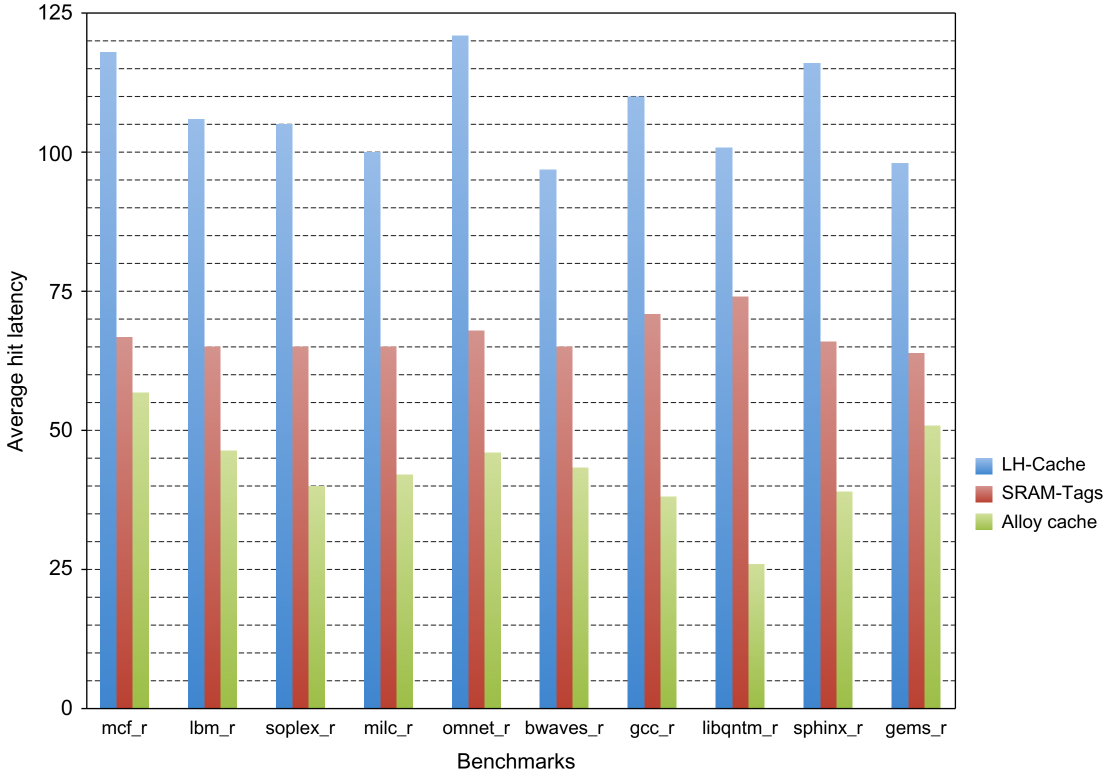


Figure 2.16 Average hit time latency in clock cycles for the L-H scheme, a currently-impractical scheme using SRAM for the tags, and the alloy cache organization. In the SRAM case, we assume the SRAM is accessible in the same time as L3 and that it is checked before L4 is accessed. The average hit latencies are 43 (alloy cache), 67 (SRAM tags), and 107 (L-H). The 10 SPECCPU2006 benchmarks used here are the most memory-intensive ones; each of them would run twice as fast if L3 were perfect.

slower). If we could speed up the miss detection, we could reduce the miss time. Two different solutions have been proposed to solve this problem: one uses a map that keeps track of the blocks in the cache (not the location of the block, just whether it is present); the other uses a memory access predictor that predicts likely misses using history prediction techniques, similar to those used for global branch prediction (see the next chapter). It appears that a small predictor can predict likely misses with high accuracy, leading to an overall lower miss penalty.

Figure 2.17 shows the speedup obtained on SPECrate for the memory-intensive benchmarks used in Figure 2.16. The alloy cache approach outperforms the LH scheme and even the impractical SRAM tags, because the combination of a fast access time for the miss predictor and good prediction results lead to a shorter time to predict a miss, and thus a lower miss penalty. The alloy cache performs close to the Ideal case, an L4 with perfect miss prediction and minimal hit time.

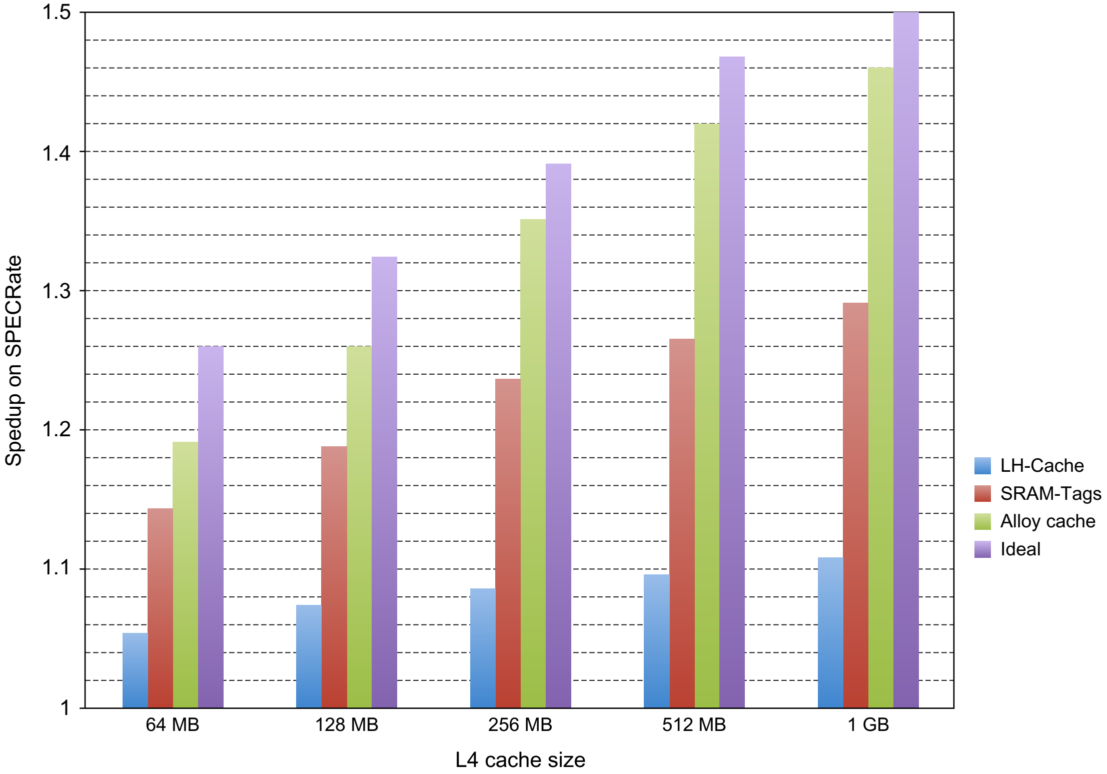


Figure 2.17 Performance speedup running the SPECrate benchmark for the LH scheme, an SRAM tag scheme, and an ideal L4 (Ideal); a speedup of 1 indicates no improvement with the L4 cache, and a speedup of 2 would be achievable if L4 were perfect and took no access time. The 10 memory-intensive benchmarks are used with each benchmark run eight times. The accompanying miss prediction scheme is used. The Ideal case assumes that only the 64-byte block requested in L4 needs to be accessed and transferred and that prediction accuracy for L4 is perfect (i.e., all misses are known at zero cost).

HBM is likely to have widespread use in a variety of different configurations, from containing the entire memory system for some high-performance, special-purpose systems to use as an L4 cache for larger server configurations.

Cache Optimization Summary

The techniques to improve hit time, bandwidth, miss penalty, and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure 2.18 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined & banked caches	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs
HBM as additional level of cache		+/–	–	+	+	3	Depends on new packaging technology. Effects depend heavily on hit rate improvements

Figure 2.18 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. Generally, no technique helps more than one category.

2.4 Virtual Memory and Virtual Machines

A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM)… a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

Gerald Popek and Robert Goldberg,
"Formal requirements for virtualizable third generation architectures,"
Communications of the ACM (July 1974).

Section B.4 in Appendix B describes the key concepts in virtual memory. Recall that virtual memory allows the physical memory to be treated as a cache of secondary storage (which may be either disk or solid state). Virtual memory moves pages between the two levels of the memory hierarchy, just as caches move blocks between levels. Likewise, TLBs act as caches on the page table, eliminating the need to do a memory access every time an address is translated. Virtual memory also provides separation between processes that share one physical memory but have separate virtual address spaces. Readers should ensure that they understand both functions of virtual memory before continuing.

In this section, we focus on additional issues in protection and privacy between processes sharing the same processor. Security and privacy are two of the most vexing challenges for information technology in 2017. Electronic burglaries, often involving lists of credit card numbers, are announced regularly, and it's widely believed that many more go unreported. Of course, such problems arise from programming errors that allow a cyberattack to access data it should be unable to access. Programming errors are a fact of life, and with modern complex software systems, they occur with significant regularity. Therefore both researchers and practitioners are looking for improved ways to make computing systems more secure. Although protecting information is not limited to hardware, in our view real security and privacy will likely involve innovation in computer architecture as well as in systems software.

This section starts with a review of the architecture support for protecting processes from each other via virtual memory. It then describes the added protection provided by virtual machines, the architecture requirements of virtual machines, and the performance of a virtual machine. As we will see in Chapter 6, virtual machines are a foundational technology for cloud computing.

Protection via Virtual Memory

Page-based virtual memory, including a TLB that caches page table entries, is the primary mechanism that protects processes from each other. Sections B.4 and B.5 in Appendix B review virtual memory, including a detailed description of protection via segmentation and paging in the 80x86. This section acts as a quick review; if it's too quick, please refer to the denoted Appendix B sections.

Multiprogramming, where several programs running concurrently share a computer, has led to demands for protection and sharing among programs and to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. At any instant, it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

The operating system and architecture join forces to allow processes to share the hardware yet not interfere with each other. To do this, the architecture must limit what a process can access when running a user process yet allow an operating system process to access more. At a minimum, the architecture must do the following:

1. Provide at least two modes, indicating whether the running process is a user process or an operating system process. This latter process is sometimes called a *kernel* process or a *supervisor* process.
2. Provide a portion of the processor state that a user process can use but not write. This state includes a user/supervisor mode bit, an exception enable/disable bit, and memory protection information. Users are prevented from writing this state because the operating system cannot control user processes if users can give themselves supervisor privileges, disable exceptions, or change memory protection.
3. Provide mechanisms whereby the processor can go from user mode to supervisor mode and vice versa. The first direction is typically accomplished by a *system call*, implemented as a special instruction that transfers control to a dedicated location in supervisor code space. The PC is saved from the point of the system call, and the processor is placed in supervisor mode. The return to user mode is like a subroutine return that restores the previous user/supervisor mode.
4. Provide mechanisms to limit memory accesses to protect the memory state of a process without having to swap the process to disk on a context switch.

Appendix A describes several memory protection schemes, but by far the most popular is adding protection restrictions to each page of virtual memory. Fixed-sized pages, typically 4 KiB, 16 KiB, or larger, are mapped from the virtual address space into physical address space via a page table. The protection restrictions are included in each page table entry. The protection restrictions might determine whether a user process can read this page, whether a user process can write to this page, and whether code can be executed from this page. In addition, a process can neither read nor write a page if it is not in the page table. Because only the OS can update the page table, the paging mechanism provides total access protection.

Paged virtual memory means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. This cost would be far too dear. The solution is to rely on the principle of locality; if the accesses have locality, then the *address translations* for the accesses must also have locality. By keeping these address translations in a special cache, a memory access rarely requires a second access to translate the address. This special address translation cache is referred to as a TLB.

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page address, protection field, valid bit, and usually a use bit and a dirty bit. The operating system changes these bits by changing the value in the page table and then invalidating the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Assuming the computer faithfully obeys the restrictions on pages and maps virtual addresses to physical addresses, it would seem that we are done. Newspaper headlines suggest otherwise.

The reason we're not done is that we depend on the accuracy of the operating system as well as the hardware. Today's operating systems consist of tens of millions of lines of code. Because bugs are measured in number per thousand lines of code, there are thousands of bugs in production operating systems. Flaws in the OS have led to vulnerabilities that are routinely exploited.

This problem and the possibility that not enforcing protection could be much more costly than in the past have led some to look for a protection model with a much smaller code base than the full OS, such as virtual machines.

Protection via Virtual Machines

An idea related to virtual memory that is almost as old are virtual machines (VMs). They were first developed in the late 1960s, and they have remained an important part of mainframe computing over the years. Although largely ignored in the domain of single-user computers in the 1980s and 1990s, they have recently gained popularity because of

- the increasing importance of isolation and security in modern systems;
- the failures in security and reliability of standard operating systems;
- the sharing of a single computer among many unrelated users, such as in a data center or cloud; and
- the dramatic increases in the raw speed of processors, which make the overhead of VMs more acceptable.

The broadest definition of VMs includes basically all emulation methods that provide a standard software interface, such as the Java VM. We are interested in

VMs that provide a complete system-level environment at the binary instruction set architecture (ISA) level. Most often, the VM supports the same ISA as the underlying hardware; however, it is also possible to support a different ISA, and such approaches are often employed when migrating between ISAs in order to allow software from the departing ISA to be used until it can be ported to the new ISA. Our focus here will be on VMs where the ISA presented by the VM and the underlying hardware match. Such VMs are called (operating) *system virtual machines*. IBM VM/370, VMware ESX Server, and Xen are examples. They present the illusion that the users of a VM have an entire computer to themselves, including a copy of the operating system. A single computer runs multiple VMs and can support a number of different operating systems (OSes). On a conventional platform, a single OS "owns" all the hardware resources, but with a VM, multiple OSes all share the hardware resources.

The software that supports VMs is called a *virtual machine monitor* (VMM) or *hypervisor*; the VMM is the heart of virtual machine technology. The underlying hardware platform is called the *host*, and its resources are shared among the *guest* VMs. The VMM determines how to map virtual resources to physical resources: A physical resource may be time-shared, partitioned, or even emulated in software. The VMM is much smaller than a traditional OS; the isolation portion of a VMM is perhaps only 10,000 lines of code.

In general, the cost of processor virtualization depends on the workload. User-level processor-bound programs, such as SPECCPU2006, have zero virtualization overhead because the OS is rarely invoked, so everything runs at native speeds. Conversely, I/O-intensive workloads generally are also OS-intensive and execute many system calls (which doing I/O requires) and privileged instructions that can result in high virtualization overhead. The overhead is determined by the number of instructions that must be emulated by the VMM and how slowly they are emulated. Therefore, when the guest VMs run the same ISA as the host, as we assume here, the goal of the architecture and the VMM is to run almost all instructions directly on the native hardware. On the other hand, if the I/O-intensive workload is also *I/O-bound*, the cost of processor virtualization can be completely hidden by low processor utilization because it is often waiting for I/O.

Although our interest here is in VMs for improving protection, VMs provide two other benefits that are commercially significant:

1. *Managing software*—VMs provide an abstraction that can run the complete software stack, even including old operating systems such as DOS. A typical deployment might be some VMs running legacy OSes, many running the current stable OS release, and a few testing the next OS release.
2. *Managing hardware*—One reason for multiple servers is to have each application running with its own compatible version of the operating system on separate computers, as this separation can improve dependability. VMs allow these separate software stacks to run independently yet share hardware, thereby consolidating the number of servers. Another example is that most newer VMMs support migration of a running VM to a different computer, either to

balance load or to evacuate from failing hardware. The rise of cloud computing has made the ability to swap out an entire VM to another physical processor increasingly useful.

These two reasons are why cloud-based servers, such as Amazon's, rely on virtual machines.

Requirements of a Virtual Machine Monitor

What must a VM monitor do? It presents a software interface to guest software, it must isolate the state of guests from each other, and it must protect itself from guest software (including guest OSes). The qualitative requirements are

- Guest software should behave on a VM exactly as if it were running on the native hardware, except for performance-related behavior or limitations of fixed resources shared by multiple VMs.
- Guest software should not be able to directly change allocation of real system resources.

To "virtualize" the processor, the VMM must control just about everything—access to privileged state, address translation, I/O, exceptions and interrupts—even though the guest VM and OS currently running are temporarily using them.

For example, in the case of a timer interrupt, the VMM would suspend the currently running guest VM, save its state, handle the interrupt, determine which guest VM to run next, and then load its state. Guest VMs that rely on a timer interrupt are provided with a virtual timer and an emulated timer interrupt by the VMM.

To be in charge, the VMM must be at a higher privilege level than the guest VM, which generally runs in user mode; this also ensures that the execution of any privileged instruction will be handled by the VMM. The basic requirements of system virtual machines are almost identical to those for the previously mentioned paged virtual memory:

- At least two processor modes, system and user.
- A privileged subset of instructions that is available only in system mode, resulting in a trap if executed in user mode. All system resources must be controllable only via these instructions.

Instruction Set Architecture Support for Virtual Machines

If VMs are planned for during the design of the ISA, it's relatively easy to reduce both the number of instructions that must be executed by a VMM and how long it takes to emulate them. An architecture that allows the VM to execute directly on the hardware earns the title *virtualizable*, and the IBM 370 architecture proudly bears that label.

However, because VMs have been considered for desktop and PC-based server applications only fairly recently, most instruction sets were created without virtualization in mind. These culprits include 80x86 and most of the original RISC architectures, although the latter had fewer issues than the 80x86 architecture. Recent additions to the x86 architecture have attempted to remedy the earlier shortcomings, and RISC V explicitly includes support for virtualization.

Because the VMM must ensure that the guest system interacts only with virtual resources, a conventional guest OS runs as a user mode program on top of the VMM. Then, if a guest OS attempts to access or modify information related to hardware resources via a privileged instruction—for example, reading or writing the page table pointer—it will trap to the VMM. The VMM can then effect the appropriate changes to corresponding real resources.

Therefore, if any instruction that tries to read or write such sensitive information traps when executed in user mode, the VMM can intercept it and support a virtual version of the sensitive information as the guest OS expects.

In the absence of such support, other measures must be taken. A VMM must take special precautions to locate all problematic instructions and ensure that they behave correctly when executed by a guest OS, thereby increasing the complexity of the VMM and reducing the performance of running the VM. Sections 2.5 and 2.7 give concrete examples of problematic instructions in the 80x86 architecture. One attractive extension allows the VM and the OS to operate at different privilege levels, each of which is distinct from the user level. By introducing an additional privilege level, some OS operations—e.g., those that exceed the permissions granted to a user program but do not require intervention by the VMM (because they cannot affect any other VM)—can execute directly without the overhead of trapping and invoking the VMM. The Xen design, which we examine shortly, makes use of three privilege levels.

Impact of Virtual Machines on Virtual Memory and I/O

Another challenge is virtualization of virtual memory, as each guest OS in every VM manages its own set of page tables. To make this work, the VMM separates the notions of *real* and *physical memory* (which are often treated synonymously) and makes real memory a separate, intermediate level between virtual memory and physical memory. (Some use the terms *virtual memory, physical memory*, and *machine memory* to name the same three levels.) The guest OS maps virtual memory to real memory via its page tables, and the VMM page tables map the guests' real memory to physical memory. The virtual memory architecture is specified either via page tables, as in IBM VM/370 and the 80x86, or via the TLB structure, as in many RISC architectures.

Rather than pay an extra level of indirection on every memory access, the VMM maintains a *shadow page table* that maps directly from the guest virtual address space to the physical address space of the hardware. By detecting all modifications to the guest's page table, the VMM can ensure that the shadow page table entries being used by the hardware for translations correspond to those of the guest OS environment, with the exception of the correct physical pages substituted for the real pages in the guest tables. Therefore the VMM must trap any attempt by the guest OS to change its page table or to access the page table pointer. This is commonly done by write protecting the guest page tables and trapping any access to the page table pointer by a guest OS. As previously noted, the latter happens naturally if accessing the page table pointer is a privileged operation.

The IBM 370 architecture solved the page table problem in the 1970s with an additional level of indirection that is managed by the VMM. The guest OS keeps its page tables as before, so the shadow pages are unnecessary. AMD has implemented a similar scheme for its 80x86.

To virtualize the TLB in many RISC computers, the VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM. To pull this off, any instructions that access the TLB must trap. TLBs with Process ID tags can support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch. Meanwhile, in the background, the VMM supports a mapping between the VMs' virtual Process IDs and the real Process IDs. Section L.7 of online Appendix L describes additional details.

The final portion of the architecture to virtualize is I/O. This is by far the most difficult part of system virtualization because of the increasing number of I/O devices attached to the computer *and* the increasing diversity of I/O device types. Another difficulty is the sharing of a real device among multiple VMs, and yet another comes from supporting the myriad of device drivers that are required, especially if different guest OSes are supported on the same VM system. The VM illusion can be maintained by giving each VM generic versions of each type of I/O device driver, and then leaving it to the VMM to handle real I/O.

The method for mapping a virtual-to-physical I/O device depends on the type of device. For example, physical disks are normally partitioned by the VMM to create virtual disks for guest VMs, and the VMM maintains the mapping of virtual tracks and sectors to the physical ones. Network interfaces are often shared between VMs in very short time slices, and the job of the VMM is to keep track of messages for the virtual network addresses to ensure that guest VMs receive only messages intended for them.

Extending the Instruction Set for Efficient Virtualization and Better Security

In the past 5–10 years, processor designers, including those at AMD and Intel (and to a lesser extent ARM), have introduced instruction set extensions to more efficiently support virtualization. Two primary areas of performance improvement have been in handling page tables and TLBs (the cornerstone of virtual memory) and in I/O, specifically handling interrupts and DMA. Virtual memory performance is enhanced by avoiding unnecessary TLB flushes and by using the nested page table mechanism, employed by IBM decades earlier, rather than a complete set of shadow page tables (see Section L.7 in Appendix L). To improve I/O performance, architectural extensions are added that allow a device to directly use DMA to move data (eliminating a potential copy by the VMM) and allow device interrupts and commands to be handled by the guest OS directly. These extensions show significant performance gains in applications that are intensive either in their memory-management aspects or in the use of I/O.

With the broad adoption of public cloud systems for running critical applications, concerns have risen about security of data in such applications. Any malicious code that is able to access a higher privilege level than data that must be kept secure compromises the system. For example, if you are running a credit card processing application, you must be absolutely certain that malicious users cannot get access to the credit card numbers, even when they are using the same hardware and intentionally attack the OS or even the VMM. Through the use of virtualization, we can prevent accesses by an outside user to the data in a different VM, and this provides significant protection compared to a multiprogrammed environment. That might not be enough, however, if the attacker compromises the VMM or can find out information by observations in another VMM. For example, suppose the attacker penetrates the VMM; the attacker can then remap memory so as to access any portion of the data.

Alternatively, an attack might rely on a Trojan horse (see Appendix B) introduced into the code that can access the credit cards. Because the Trojan horse is running in the same VM as the credit card processing application, the Trojan horse only needs to exploit an OS flaw to gain access to the critical data. Most cyberattacks have used some form of Trojan horse, typically exploiting an OS flaw, that either has the effect of returning access to the attacker while leaving the CPU still in privilege mode or allows the attacker to upload and execute code as if it were part of the OS. In either case, the attacker obtains control of the CPU and, using the higher privilege mode, can proceed to access anything within the VM. Note that encryption alone does not prevent this attacker. If the data in memory is unencrypted, which is typical, then the attacker has access to all such data. Furthermore, if the attacker knows where the encryption key is stored, the attacker can freely access the key and then access any encrypted data.

More recently, Intel introduced a set of instruction set extensions, called the software guard extensions (SGX), to allow user programs to create *enclaves*, portions of code and data that are always encrypted and decrypted only on use and only with the key provided by the user code. Because the enclave is always encrypted, standard OS operations for virtual memory or I/O can access the enclave (e.g., to move a page) but cannot extract any information. For an enclave to work, all the code and all the data required must be part of the enclave. Although the topic of finer-grained protection has been around for decades, it has gotten little traction before because of the high overhead and because other solutions that are more efficient and less intrusive have been acceptable. The rise of cyberattacks and the amount of confidential information online have led to a reexamination of techniques for improving such fine-grained security. Like Intel's SGX, IBM and AMD's recent processors support on-the-fly encryption of memory.

An Example VMM: The Xen Virtual Machine

Early in the development of VMs, a number of inefficiencies became apparent. For example, a guest OS manages its virtual-to-real page mapping, but this mapping is ignored by the VMM, which performs the actual mapping to physical pages. In other words, a significant amount of wasted effort is expended just to keep the guest OS happy. To reduce such inefficiencies, VMM developers decided that it may be worthwhile to allow the guest OS to be aware that it is running on a VM. For example, a guest OS could assume a real memory as large as its virtual memory so that no memory management is required by the guest OS.

Allowing small modifications to the guest OS to simplify virtualization is referred to as *paravirtualization*, and the open source Xen VMM is a good example. The Xen VMM, which is used in Amazon's web services data centers, provides a guest OS with a virtual machine abstraction that is similar to the physical hardware, but drops many of the troublesome pieces. For example, to avoid flushing the TLB, Xen maps itself into the upper 64 MiB of the address space of each VM. Xen allows the guest OS to allocate pages, checking only to be sure the guest OS does not violate protection restrictions. To protect the guest OS from the user programs in the VM, Xen takes advantage of the four protection levels available in the 80x86. The Xen VMM runs at the highest privilege level (0), the guest OS runs at the next level (1), and the applications run at the lowest privilege level (3). Most OSes for the 80x86 keep everything at privilege levels 0 or 3.

For subsetting to work properly, Xen modifies the guest OS to not use problematic portions of the architecture. For example, the port of Linux to Xen changes about 3000 lines, or about 1% of the 80x86-specific code. These changes, however, do not affect the application binary interfaces of the guest OS.

To simplify the I/O challenge of VMs, Xen assigned privileged virtual machines to each hardware I/O device. These special VMs are called *driver domains*. (Xen calls VMs "domains.") Driver domains run the physical device drivers, although interrupts are still handled by the VMM before being sent to the appropriate driver domain. Regular VMs, called *guest domains*, run simple virtual device drivers that must communicate with the physical device drivers in the driver domains over a channel to access the physical I/O hardware. Data are sent between guest and driver domains by page remapping.

2.5 Cross-Cutting Issues: The Design of Memory Hierarchies

This section describes four topics discussed in other chapters that are fundamental to memory hierarchies.

Protection, Virtualization, and Instruction Set Architecture

Protection is a joint effort of architecture and operating systems, but architects had to modify some awkward details of existing instruction set architectures when virtual memory became popular. For example, to support virtual memory in the IBM

370, architects had to change the successful IBM 360 instruction set architecture that had been announced just 6 years before. Similar adjustments are being made today to accommodate virtual machines.

For example, the 80x86 instruction `POPF` loads the flag registers from the top of the stack in memory. One of the flags is the Interrupt Enable (IE) flag. Until recent changes to support virtualization, running the `POPF` instruction in user mode, rather than trapping it, simply changed all the flags except IE. In system mode, it does change the IE flag. Because a guest OS runs in user mode inside a VM, this was a problem, as the OS would expect to see a changed IE. Extensions of the 80x86 architecture to support virtualization eliminated this problem.

Historically, IBM mainframe hardware and VMM took three steps to improve performance of virtual machines:

1. Reduce the cost of processor virtualization.
2. Reduce interrupt overhead cost due to the virtualization.
3. Reduce interrupt cost by steering interrupts to the proper VM without invoking VMM.

IBM is still the gold standard of virtual machine technology. For example, an IBM mainframe ran thousands of Linux VMs in 2000, while Xen ran 25 VMs in 2004 (Clark et al., 2004). Recent versions of Intel and AMD chipsets have added special instructions to support devices in a VM to mask interrupts at lower levels from each VM and to steer interrupts to the appropriate VM.

Autonomous Instruction Fetch Units

Many processors with out-of-order execution and even some with simply deep pipelines decouple the instruction fetch (and sometimes initial decode), using a separate instruction fetch unit (see Chapter 3). Typically, the instruction fetch unit accesses the instruction cache to fetch an entire block before decoding it into individual instructions; such a technique is particularly useful when the instruction length varies. Because the instruction cache is accessed in blocks, it no longer makes sense to compare miss rates to processors that access the instruction cache once per instruction. In addition, the instruction fetch unit may prefetch blocks into the L1 cache; these prefetches may generate additional misses, but may actually reduce the total miss penalty incurred. Many processors also include data prefetching, which may increase the data cache miss rate, even while decreasing the total data cache miss penalty.

Speculation and Memory Access

One of the major techniques used in advanced pipelines is speculation, whereby an instruction is tentatively executed before the processor knows whether it is really needed. Such techniques rely on branch prediction, which if incorrect requires that the speculated instructions are flushed from the pipeline. There are two separate issues in a memory system supporting speculation: protection and performance. With speculation, the processor may generate memory references, which will never be used because the instructions were the result of incorrect speculation. Those references, if executed, could generate protection exceptions. Obviously, such faults should occur only if the instruction is actually executed. In the next chapter, we will see how such "speculative exceptions" are resolved. Because a speculative processor may generate accesses to both the instruction and data caches, and subsequently not use the results of those accesses, speculation may increase the cache miss rates. As with prefetching, however, such speculation may actually lower the total cache miss penalty. The use of speculation, like the use of prefetching, makes it misleading to compare miss rates to those seen in processors without speculation, even when the ISA and cache structures are otherwise identical.

Special Instruction Caches

One of the biggest challenges in superscalar processors is to supply the instruction bandwidth. For designs that translate the instructions into micro-operations, such as most recent Arm and i7 processors, instruction bandwidth demands and branch misprediction penalties can be reduced by keeping a small cache of recently translated instructions. We explore this technique in greater depth in the next chapter.

Coherency of Cached Data

Data can be found in memory and in the cache. As long as the processor is the sole component changing or reading the data and the cache stands between the processor and memory, there is little danger in the processor seeing the old or *stale* copy. As we will see, multiple processors and I/O devices raise the opportunity for copies to be inconsistent and to read the wrong copy.

The frequency of the cache coherency problem is different for multiprocessors than for I/O. Multiple data copies are a rare event for I/O—one to be avoided whenever possible—but a program running on multiple processors will *want* to have copies of the same data in several caches. Performance of a multiprocessor program depends on the performance of the system when sharing data.

The *I/O cache coherency* question is this: where does the I/O occur in the computer—between the I/O device and the cache or between the I/O device and main memory? If input puts data into the cache and output reads data from the cache, both I/O and the processor see the same data. The difficulty in this approach is that it interferes with the processor and can cause the processor to stall for I/O. Input may also interfere with the cache by displacing some information with new data that are unlikely to be accessed soon.

The goal for the I/O system in a computer with a cache is to prevent the stale data problem while interfering as little as possible. Many systems therefore prefer that I/O occur directly to main memory, with main memory acting as an I/O buffer. If a write-through cache were used, then memory would have an up-to-date copy of the information, and there would be no stale data issue for output. (This benefit is a reason processors used write through.) However, today write through is usually found only in first-level data caches backed by an L2 cache that uses write back.

Input requires some extra work. The software solution is to guarantee that no blocks of the input buffer are in the cache. A page containing the buffer can be marked as noncachable, and the operating system can always input to such a page. Alternatively, the operating system can flush the buffer addresses from the cache before the input occurs. A hardware solution is to check the I/O addresses on input to see if they are in the cache. If there is a match of I/O addresses in the cache, the cache entries are invalidated to avoid stale data. All of these approaches can also be used for output with write-back caches.

Processor cache coherency is a critical subject in the age of multicore processors, and we will examine it in detail in Chapter 5.

2.6 Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700

This section reveals the ARM Cortex-A53 (hereafter called the A53) and Intel Core i76700 (hereafter called i7) memory hierarchies and shows the performance of their components on a set of single-threaded benchmarks. We examine the Cortex-A53 first because it has a simpler memory system; we go into more detail for the i7, tracing out a memory reference in detail. This section presumes that readers are familiar with the organization of a two-level cache hierarchy using virtually indexed caches. The basics of such a memory system are explained in detail in Appendix B, and readers who are uncertain of the organization of such a system are strongly advised to review the Opteron example in Appendix B. Once they understand the organization of the Opteron, the brief explanation of the A53 system, which is similar, will be easy to follow.

The ARM Cortex-A53

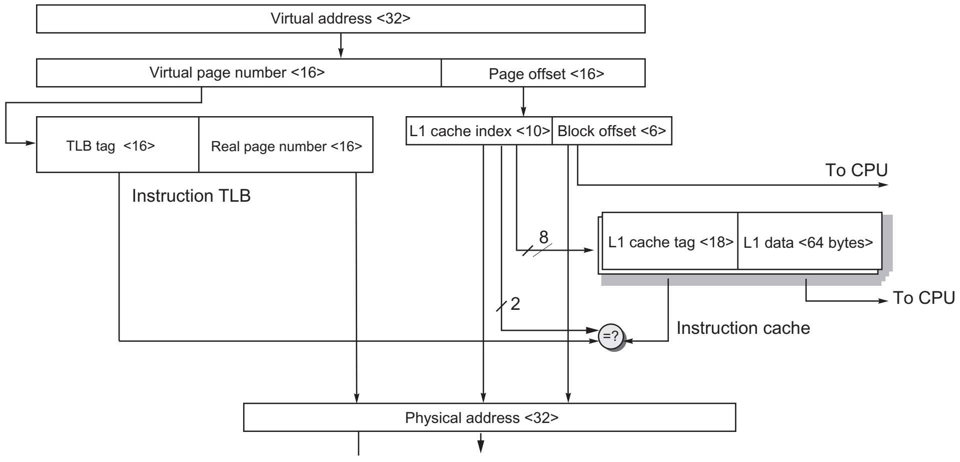
The Cortex-A53 is a configurable core that supports the ARMv8A instruction set architecture, which includes both 32-bit and 64-bit modes. The Cortex-A53 is delivered as an IP (intellectual property) core. IP cores are the dominant form of technology delivery in the embedded, PMD, and related markets; billions of ARM and MIPS processors have been created from these IP cores. Note that IP cores are different from the cores in the Intel i7 or AMD Athlon multicores. An IP core (which may itself be a multicore) is designed to be incorporated with other logic (thus it is the core of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application. For example, the Cortex-A53 IP core is used in a variety of tablets and smartphones; it is designed to be highly energy-efficient, a key criteria in battery-based PMDs. The A53 core is capable of being configured with multiple cores per chip for use in high-end PMDs; our discussion here focuses on a single core.

Generally, IP cores come in two flavors. *Hard cores* are optimized for a particular semiconductor vendor and are black boxes with external (but still on-chip) interfaces. Hard cores typically allow parametrization only of logic outside the core, such as L2 cache sizes, and the IP core cannot be modified. *Soft cores* are usually delivered in a form that uses a standard library of logic elements. A soft core can be compiled for different semiconductor vendors and can also be modified, although extensive modifications are very difficult because of the complexity of modern-day IP cores. In general, hard cores provide higher performance and smaller die area, while soft cores allow retargeting to other vendors and can be more easily modified.

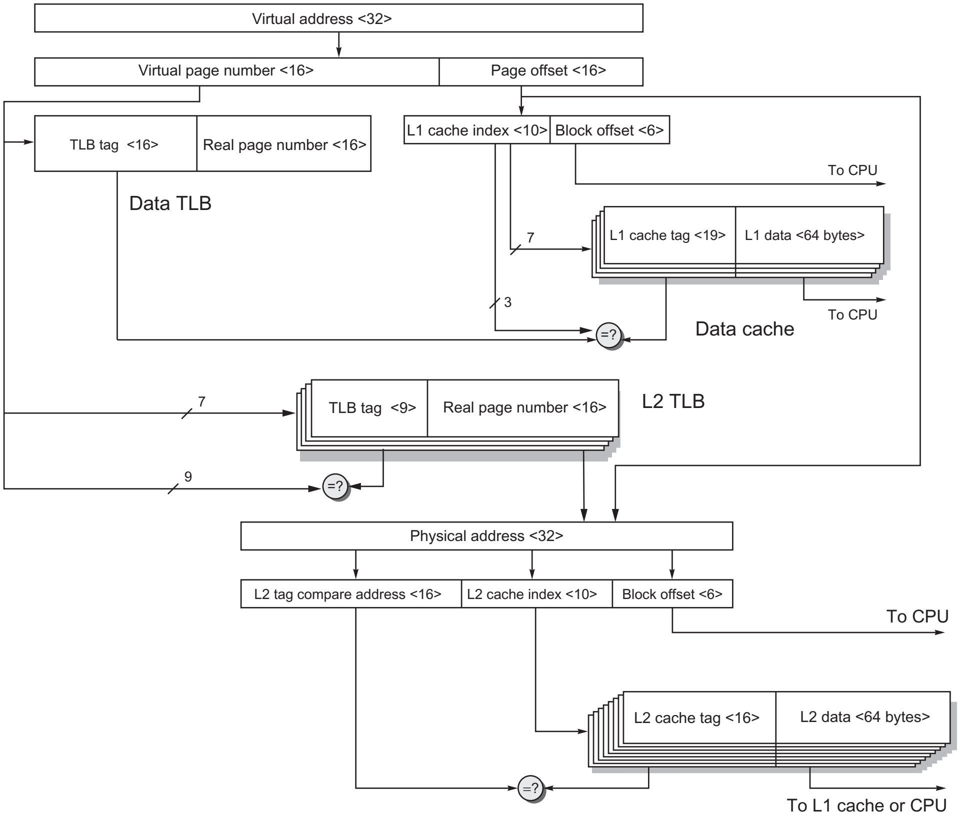
The Cortex-A53 can issue two instructions per clock at clock rates up to 1.3 GHz. It supports both a two-level TLB and a two-level cache; Figure 2.19 summarizes the organization of the memory hierarchy. The critical term is returned first, and the processor can continue while the miss completes; a memory system with up to four banks can be supported. For a D-cache of 32 KiB and a page size of 4 KiB, each physical page could map to two different cache addresses; such aliases are avoided by hardware detection on a miss as in Section B.3 of Appendix B. Figure 2.20 shows how the 32-bit virtual address is used to index the TLB and the caches, assuming 32 KiB primary caches and a 1 MiB secondary cache with 16 KiB page size.

Structure	Size	Organization	Typical miss penalty (clock cycles)
Instruction MicroTLB	10 entries	Fully associative	2
Data MicroTLB	10 entries	Fully associative	2
L2 Unified TLB	512 entries	4-way set associative	20
L1 Instruction cache	8–64 KiB	2-way set associative; 64-byte block	13
L1 Data cache	8–64 KiB	2-way set associative; 64-byte block	13
L2 Unified cache	128 KiB to 2 MiB	16-way set associative; LRU	124

Figure 2.19 The memory hierarchy of the Cortex A53 includes multilevel TLBs and caches. A page map cache keeps track of the location of a physical page for a set of virtual pages; it reduces the L2 TLB miss penalty. The L1 caches are virtually indexed and physically tagged; both the L1 D cache and L2 use a write-back policy defaulting to allocate on write. Replacement policy is LRU approximation in all the caches. Miss penalties to L2 are higher if both a MicroTLB and L1 miss occur. The L2 to main memory bus is 64–128 bits wide, and the miss penalty is larger for the narrow bus.


Virtual address <32>
Virtual page number <16>
Page offset <16>
TLB tag <16>
Real page number <16>
L1 cache index <10>
Block offset <6>
To CPU
Instruction TLB
8
L1 cache tag <18>
L1 data <64 bytes>
2
To CPU
=?
Instruction cache
Physical address <32>
To L2 (see part b below)

(A) **The instruction access path**


Virtual address <32>
Virtual page number <16>
Page offset <16>
TLB tag <16>
Real page number <16>
L1 cache index <10>
Block offset <6>
To CPU
Data TLB
7
L1 cache tag <19>
L1 data <64 bytes>
3
To CPU
=?
Data cache
7
TLB tag <9>
Real page number <16>
L2 TLB
9
=?
Physical address <32>
L2 tag compare address <16>
L2 cache index <10>
Block offset <6>
To CPU
L2 cache tag <16>
L2 data <64 bytes>
=?
To L1 cache or CPU

(B) **The data access path**

Figure 2.20 The virtual address, physical and data blocks for the ARM Cortex-A53 caches and TLBs, assuming 32-bit addresses. The top half (A) shows the instruction access; the bottom half (B) shows the data access, including L2. The TLB (instruction or data) is fully associative each with 10 entries, using a 64 KiB page in this example. The L1 I-cache is two-way set associative, with 64-byte blocks and 32 KiB capacity; the L1 D-cache is 32 KiB, four-way set associative, and 64-byte blocks. The L2 TLB is 512 entries and four-way set associative. The L2 cache is 16-way set associative with 64-byte blocks and 128 cKiB to 2 MiB capacity; a 1 MiB L2 is shown. This figure doesn't show the valid bits and protection bits for the caches and TLB.

Performance of the Cortex-A53 Memory Hierarchy

The memory hierarchy of the Cortex-A8 was measured with 32 KiB primary caches and a 1 MiB L2 cache running the SPECInt2006 benchmarks. The instruction cache miss rates for these SPECInt2006 are very small even for just the L1: close to zero for most and under 1% for all of them. This low rate probably results from the computationally intensive nature of the SPECCPU programs and the two-way set associative cache that eliminates most conflict misses.

Figure 2.21 shows the data cache results, which have significant L1 and L2 miss rates. The L1 rate varies by a factor of 75, from 0.5% to 37.3% with a median miss rate of 2.4%. The global L2 miss rate varies by a factor of 180, from 0.05% to 9.0% with a median of 0.3%. MCF, which is known as a cache buster, sets the upper bound and significantly affects the mean. Remember that the L2 global miss rate is significantly lower than the L2 local miss rate; for example, the median L2 stand-alone miss rate is 15.1% versus the global miss rate of 0.3%.

Using these miss penalties in Figure 2.19, Figure 2.22 shows the average penalty per data access. Although the L1 miss rates are about seven times higher than the L2 miss rate, the L2 penalty is 9.5 times as high, leading to L2 misses slightly dominating for the benchmarks that stress the memory system. In the next chapter, we will examine the impact of the cache misses on overall CPI.

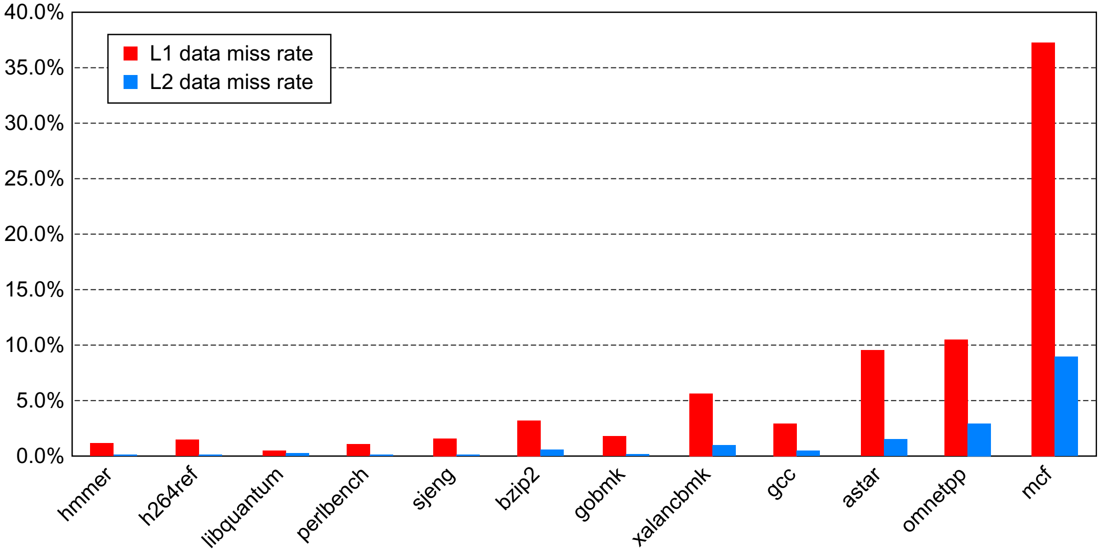


Figure 2.21 The data miss rate for ARM with a 32 KiB L1 and the global data miss rate for a 1 MiB L2 using the SPECInt2006 benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate that is counting all references, including those that hit in L1. MCF is known as a cache buster.

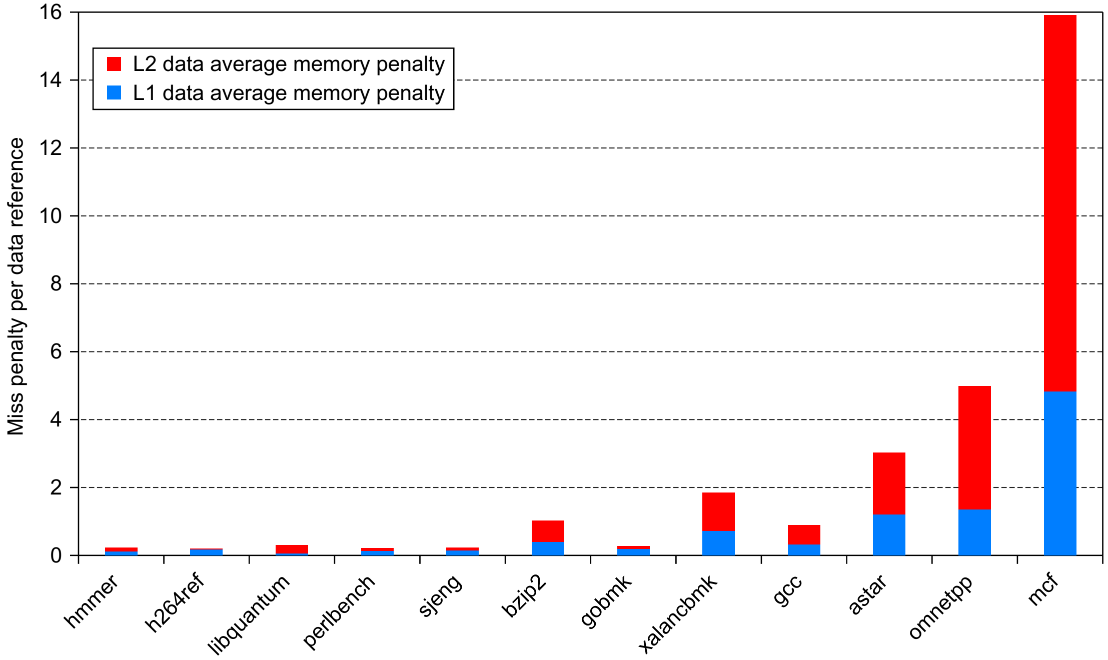


Figure 2.22 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the A53 processor when running SPECInt2006. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

The Intel Core i7 6700

The i7 supports the x86-64 instruction set architecture, a 64-bit extension of the 80x86 architecture. The i7 is an out-of-order execution processor that includes four cores. In this chapter, we focus on the memory system design and performance from the viewpoint of a single core. The system performance of multiprocessor designs, including the i7 multicore, is examined in detail in Chapter 5.

Each core in an i7 can execute up to four 80x86 instructions per clock cycle, using a multiple issue, dynamically scheduled, 16-stage pipeline, which we describe in detail in Chapter 3. The i7 can also support up to two simultaneous threads per processor, using a technique called simultaneous multithreading, described in Chapter 4. In 2017 the fastest i7 had a clock rate of 4.0 GHz (in Turbo Boost mode), which yielded a peak instruction execution rate of 16 billion instructions per second, or 64 billion instructions per second for the four-core design. Of course, there is a big gap between peak and sustained performance, as we will see over the next few chapters.

The i7 can support up to three memory channels, each consisting of a separate set of DIMMs, and each of which can transfer in parallel. Using DDR3-1066 (DIMM PC8500), the i7 has a peak memory bandwidth of just over 25 GB/s.

i7 uses 48-bit virtual addresses and 36-bit physical addresses, yielding a maximum physical memory of 36 GiB. Memory management is handled with a two-level TLB (see Appendix B, Section B.4), summarized in Figure 2.23.

Figure 2.24 summarizes the i7's three-level cache hierarchy. The first-level caches are virtually indexed and physically tagged (see Appendix B, Section B.3), while the L2 and L3 caches are physically indexed. Some versions of the i7 6700 will support a fourth-level cache using HBM packaging.

Figure 2.25 is labeled with the steps of an access to the memory hierarchy. First, the PC is sent to the instruction cache. The instruction cache index is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{32\text{K}}{64 \times 8} = 64 = 2^6$$

Characteristic	**Instruction TLB**	**Data DLB**	**Second-level TLB**
Entries	128	64	1536
Associativity	8-way	4-way	12-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access latency	1 cycle	1 cycle	8 cycles
Miss	9 cycles	9 cycles	Hundreds of cycles to access page table

Figure 2.23 Characteristics of the i7's TLB structure, which has separate first-level instruction and data TLBs, both backed by a joint second-level TLB. The first-level TLBs support the standard 4 KiB page size, as well as having a limited number of entries of large 2–4 MiB pages; only 4 KiB pages are supported in the second-level TLB. The i7 has the ability to handle two L2 TLB misses in parallel. See Section L.3 of online Appendix L for more discussion of multilevel TLBs and support for multiple page sizes.

Characteristic	**L1**	**L2**	**L3**
Size	32 KiB I/32 KiB D	256 KiB	2 MiB per core
Associativity	both 8-way	4-way	16-way
Access latency	4 cycles, pipelined	12 cycles	44 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorithm

Figure 2.24 Characteristics of the three-level cache hierarchy in the i7. All three caches use write back and a block size of 64 bytes. The L1 and L2 caches are separate for each core, whereas the L3 cache is shared among the cores on a chip and is a total of 2 MiB per core. All three caches are nonblocking and allow multiple outstanding writes. A merging write buffer is used for the L1 cache, which holds data in the event that the line is not present in L1 when it is written. (That is, an L1 write miss does not cause the line to be allocated.) L3 is inclusive of L1 and L2; we explore this property in further detail when we explain multiprocessor caches. Replacement is by a variant on pseudo-LRU; in the case of L3, the block replaced is always the lowest numbered way whose access bit is off. This is not quite random but is easy to compute.

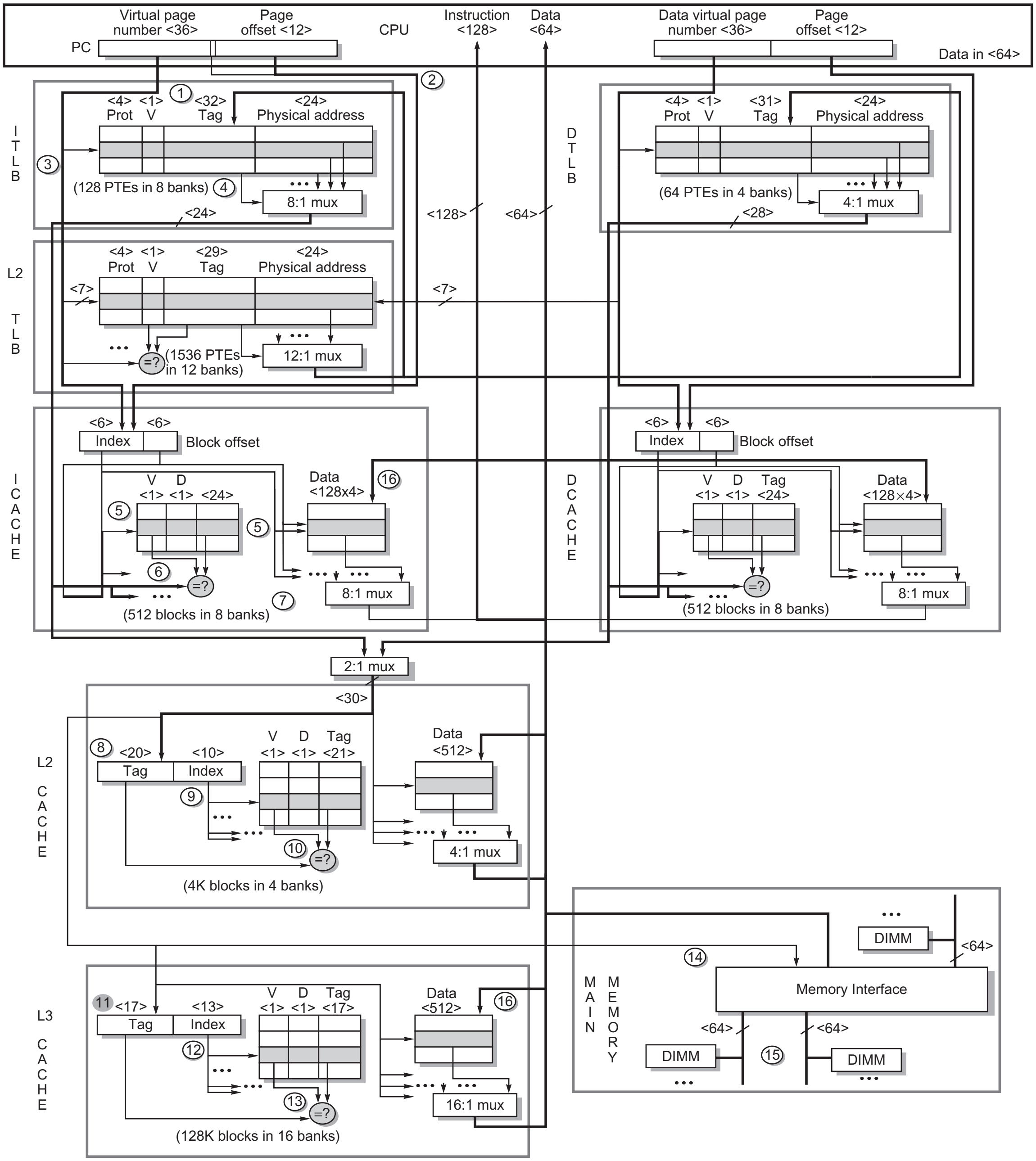

Virtual page number <36>
Page offset <12>
PC
CPU
Instruction <128>
Data <64>
Data virtual page number <36>
Page offset <12>
Data in <64>
①
②
③
I T L B
<4> Prot
<1> V
<32> Tag
<24> Physical address
(128 PTEs in 8 banks) ④
8:1 mux
<24>
<128>
<64>
D T L B
<4> Prot
<1> V
<31> Tag
<24> Physical address
(64 PTEs in 4 banks)
4:1 mux
<28>
L2 T L B
<4> Prot
<1> V
<29> Tag
<24> Physical address
<7>
<7>
=?
(1536 PTEs in 12 banks)
12:1 mux
I C A C H E
<6> Index
<6> Block offset
V D
<1> <1> <24>
Data <128x4>
⑤
⑤
⑥
⑦
⑯
=?
8:1 mux
(512 blocks in 8 banks)
D C A C H E
<6> Index
<6> Block offset
V D Tag
<1> <1> <24>
Data <128×4>
=?
8:1 mux
(512 blocks in 8 banks)
2:1 mux
<30>
L2 C A C H E
⑧
<20> Tag
<10> Index
V D Tag
<1> <1> <21>
Data <512>
⑨
⑩
=?
4:1 mux
(4K blocks in 4 banks)
L3 C A C H E
⑪
<17> Tag
<13> Index
V D Tag
<1> <1> <17>
Data <512>
⑫
⑬
⑯
=?
16:1 mux
(128K blocks in 16 banks)
M A I N M E M O R Y
⑭
Memory Interface
DIMM
<64>
<64>
<64>
DIMM
⑮
DIMM

Figure 2.25 The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads. Writes are similar, except that misses are handled by simply placing the data in a write buffer, because the L1 cache is not write-allocated.

or 6 bits. The page frame of the instruction's address ($36=48-12$ bits) is sent to the instruction TLB (step 1). At the same time, the 12-bit page offset from the virtual address is sent to the instruction cache (step 2). Notice that for the eight-way associative instruction cache, 12 bits are needed for the cache address: 6 bits to index the cache plus 6 bits of block offset for the 64-byte block, so no aliases are possible. The previous versions of the i7 used a four-way set associative I-cache, meaning that a block corresponding to a virtual address could actually be in two different places in the cache, because the corresponding physical address could have either a 0 or 1 in this location. For instructions this did not pose a problem because even if an instruction appeared in the cache in two different locations, the two versions must be the same. If such duplication, or aliasing, of data is allowed, the cache must be checked when the page map is changed, which is an infrequent event. Note that a very simple use of page coloring (see Appendix B, Section B.3) can eliminate the possibility of these aliases. If even-address virtual pages are mapped to even-address physical pages (and the same for odd pages), then these aliases can never occur because the low-order bit in the virtual and physical page number will be identical.

The instruction TLB is accessed to find a match between the address and a valid page table entry (PTE) (steps 3 and 4). In addition to translating the address, the TLB checks to see if the PTE demands that this access result in an exception because of an access violation.

An instruction TLB miss first goes to the L2 TLB, which contains 1536 PTEs of 4 KiB page sizes and is 12-way set associative. It takes 8 clock cycles to load the L1 TLB from the L2 TLB, which leads to the 9-cycle miss penalty including the initial clock cycle to access the L1 TLB. If the L2 TLB misses, a hardware algorithm is used to walk the page table and update the TLB entry. Sections L.5 and L.6 of online Appendix L describe page table walkers and page structure caches. In the worst case, the page is not in memory, and the operating system gets the page from secondary storage. Because millions of instructions could execute during a page fault, the operating system will swap in another process if one is waiting to run. Otherwise, if there is no TLB exception, the instruction cache access continues.

The index field of the address is sent to all eight banks of the instruction cache (step 5). The instruction cache tag is 36 bits − 6 bits (index) − 6 bits (block offset), or 24 bits. The four tags and valid bits are compared to the physical page frame from the instruction TLB (step 6). Because the i7 expects 16 bytes each instruction fetch, an additional 2 bits are used from the 6-bit block offset to select the appropriate 16 bytes. Therefore $6+2$ or 8 bits are used to send 16 bytes of instructions to the processor. The L1 cache is pipelined, and the latency of a hit is 4 clock cycles (step 7). A miss goes to the second-level cache.

As mentioned earlier, the instruction cache is virtually addressed and physically tagged. Because the second-level caches are physically addressed, the physical page address from the TLB is composed with the page offset to make an address to access the L2 cache. The L2 index is

$$2^{\text{Index}}=\frac{\text{Cache size}}{\text{Block size}\times\text{Set associativity}}=\frac{256\text{K}}{64\times 4}=1024=2^{10}$$

so the 30-bit block address (36-bit physical address − 6-bit block offset) is divided into a 20-bit tag and a 10-bit index (step 8). Once again, the index and tag are sent to the four banks of the unified L2 cache (step 9), which are compared in parallel. If one matches and is valid (step 10), it returns the block in sequential order after the initial 12-cycle latency at a rate of 8 bytes per clock cycle.

If the L2 cache misses, the L3 cache is accessed. For a four-core i7, which has an 8 MiB L3, the index size is

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{8\text{M}}{64 \times 16} = 8192 = 2^{13}$$

The 13-bit index (step 11) is sent to all 16 banks of the L3 (step 12). The L3 tag, which is $36 - (13 + 6) = 17$ bits, is compared against the physical address from the TLB (step 13). If a hit occurs, the block is returned after an initial latency of 42 clock cycles, at a rate of 16 bytes per clock and placed into both L1 and L3. If L3 misses, a memory access is initiated.

If the instruction is not found in the L3 cache, the on-chip memory controller must get the block from main memory. The i7 has three 64-bit memory channels that can act as one 192-bit channel, because there is only one memory controller and the same address is sent on both channels (step 14). Wide transfers happen when both channels have identical DIMMs. Each channel supports up to four DDR DIMMs (step 15). When the data return they are placed into L3 and L1 (step 16) because L3 is inclusive.

The total latency of the instruction miss that is serviced by main memory is approximately 42 processor cycles to determine that an L3 miss has occurred, plus the DRAM latency for the critical instructions. For a single-bank DDR4-2400 SDRAM and 4.0 GHz CPU, the DRAM latency is about 40 ns or 160 clock cycles to the first 16 bytes, leading to a total miss penalty of about 200 clock cycles. The memory controller fills the remainder of the 64-byte cache block at a rate of 16 bytes per I/O bus clock cycle, which takes another 5 ns or 20 clock cycles.

Because the second-level cache is a write-back cache, any miss can lead to an old block being written back to memory. The i7 has a 10-entry merging write buffer that writes back dirty cache lines when the next level in the cache is unused for a read. The write buffer is checked on a miss to see if the cache line exists in the buffer; if so, the miss is filled from the buffer. A similar buffer is used between the L1 and L2 caches. If this initial instruction is a load, the data address is sent to the data cache and data TLBs, acting very much like an instruction cache access.

Suppose the instruction is a store instead of a load. When the store issues, it does a data cache lookup just like a load. A miss causes the block to be placed in a write buffer because the L1 cache does not allocate the block on a write miss. On a hit, the store does not update the L1 (or L2) cache until later, after it is known to be nonspeculative. During this time, the store resides in a load-store queue, part of the out-of-order control mechanism of the processor.

The I7 also supports prefetching for L1 and L2 from the next level in the hierarchy. In most cases, the prefetched line is simply the next block in the cache. By prefetching only for L1 and L2, high-cost unnecessary fetches to memory are avoided.

Performance of the i7 memory system

We evaluate the performance of the i7 cache structure using the SPECint2006 benchmarks. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University. Their analysis is based on earlier work (see Prakash and Peng, 2008).

The complexity of the i7 pipeline, with its use of an autonomous instruction fetch unit, speculation, and both instruction and data prefetch, makes it hard to compare cache performance against simpler processors. As mentioned on page 110, processors that use prefetch can generate cache accesses independent of the memory accesses performed by the program. A cache access that is generated because of an actual instruction access or data access is sometimes called a *demand access* to distinguish it from a *prefetch access*. Demand accesses can come from both speculative instruction fetches and speculative data accesses, some of which are subsequently canceled (see Chapter 3 for a detailed description of speculation and instruction graduation). A speculative processor generates at least as many misses as an in-order nonspeculative processor, and typically more. In addition to demand misses, there are prefetch misses for both instructions and data.

The i7's instruction fetch unit attempts to fetch 16 bytes every cycle, which complicates comparing instruction cache miss rates because multiple instructions are fetched every cycle (roughly 4.5 on average). In fact, the entire 64-byte cache line is read and subsequent 16-byte fetches do not require additional accesses. Thus misses are tracked only on the basis of 64-byte blocks. The 32 KiB, eight-way set associative instruction cache leads to a very low instruction miss rate for the SPECint2006 programs. If, for simplicity, we measure the miss rate of SPECint2006 as the number of misses for a 64-byte block divided by the number of instructions that complete, the miss rates are all under 1% except for one benchmark (XALANCBMK), which has a 2.9% miss rate. Because a 64-byte block typically contains 16–20 instructions, the effective miss rate per instruction is much lower, depending on the degree of spatial locality in the instruction stream.

The frequency at which the instruction fetch unit is stalled waiting for the I-cache misses is similarly small (as a percentage of total cycles) increasing to 2% for two benchmarks and 12% for XALANCBMK, which has the highest I-cache miss rate. In the next chapter, we will see how stalls in the IFU contribute to overall reductions in pipeline throughput in the i7.

The L1 data cache is more interesting and even trickier to evaluate because in addition to the effects of prefetching and speculation, the L1 data cache is not write-allocated, and writes to cache blocks that are not present are not treated as misses. For this reason, we focus only on memory reads. The performance monitor measurements in the i7 separate out prefetch accesses from demand accesses, but only keep demand accesses for those instructions that graduate. The effect of speculative instructions that do not graduate is not negligible, although pipeline effects probably dominate secondary cache effects caused by speculation; we will return to the issue in the next chapter.

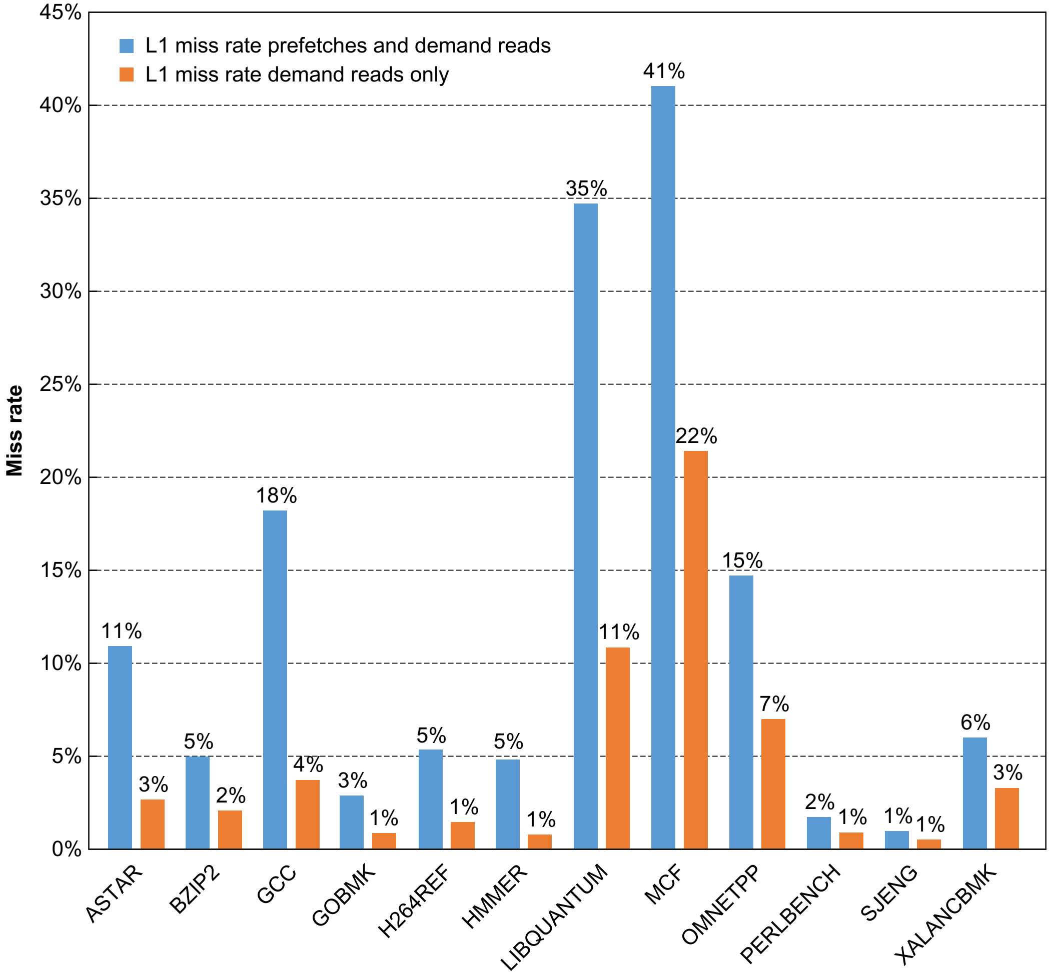


Figure 2.26 The L1 data cache miss rate for the SPECint2006 benchmarks is shown in two ways relative to the demand L1 reads: one including both demand and prefetch accesses and one including only demand accesses. The i7 separates out L1 misses for a block not present in the cache and L1 misses for a block already outstanding that is being prefetched from L2; we treat the latter group as hits because they would hit in a blocking cache. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

To address these issues, while keeping the amount of data reasonable, Figure 2.26 shows the L1 data cache misses in two ways:

1. The L1 miss rate relative to demand references given by the L1 miss rate including prefetches and speculative loads/L1 demand read references for those instructions that graduate.

2. The demand miss rate given by L1 demand misses/L1 demand read references, both measurements only for instructions that graduate.

On average, the miss rate including prefetches is 2.8 times as high as the demand-only miss rate. Comparing this data to that from the earlier i7 920, which had the same size L1, we see that the miss rate including prefetches is higher on the newer i7, but the number of demand misses, which are more likely to cause a stall, are usually fewer.

To understand the effectiveness of the aggressive prefetch mechanisms in the i7, let's look at some measurements of prefetching. Figure 2.27 shows both the fraction of L2 requests that are prefetches versus demand requests and the prefetch miss rate. The data are probably astonishing at first glance: there are roughly 1.5 times as many prefetches as there are L2 demand requests, which come directly from L1 misses. Furthermore, the prefetch miss rate is amazingly high, with an average miss rate of 58%. Although the prefetch ratio varies considerably, the prefetch miss rate is always significant. At first glance, you might conclude that the designers made a mistake: they are prefetching too much, and the miss rate is too high. Notice, however, that the benchmarks with the higher prefetch ratios (ASTAR, BZIP2, HMMER, LIBQUANTUM, and OMNETPP) also show the greatest gap between the prefetch miss rate and the demand miss rate, more than a factor of 2 in each case. The aggressive prefetching is trading prefetch misses, which occur earlier, for demand misses, which occur later; and as a result, a pipeline stall is less likely to occur due to the prefetching.

Similarly, consider the high prefetch miss rate. Suppose that the majority of the prefetches are actually useful (this is hard to measure because it involves tracking individual cache blocks), then a prefetch miss indicates a likely L2 cache miss in the future. Uncovering and handling the miss earlier via the prefetch is likely to reduce the stall cycles. Performance analysis of speculative superscalars, like the i7, has shown that cache misses tend to be the primary cause of pipeline stalls, because it is hard to keep the processor going, especially for longer running L2 and L3 misses. The Intel designers could not easily increase the size of the caches without incurring both energy and cycle time impacts; thus the use of aggressive prefetching to try to lower effective cache miss penalties is an interesting alternative approach.

With the combination of the L1 demand misses and prefetches going to L2, roughly 17% of the loads generate an L2 request. Analyzing L2 performance requires including the effects of writes (because L2 is write-allocated), as well as the prefetch hit rate and the demand hit rate. Figure 2.28 shows the miss rates of the L2 caches for demand and prefetch accesses, both versus the number of L1 references (reads and writes). As with L1, prefetches are a significant contributor, generating 75% of the L2 misses. Comparing the L2 demand miss rate with that of earlier i7 implementations (again with the same L2 size) shows that the i7 6700 has a lower L2 demand miss rate by an approximate factor of 2, which may well justify the higher prefetch miss rate.

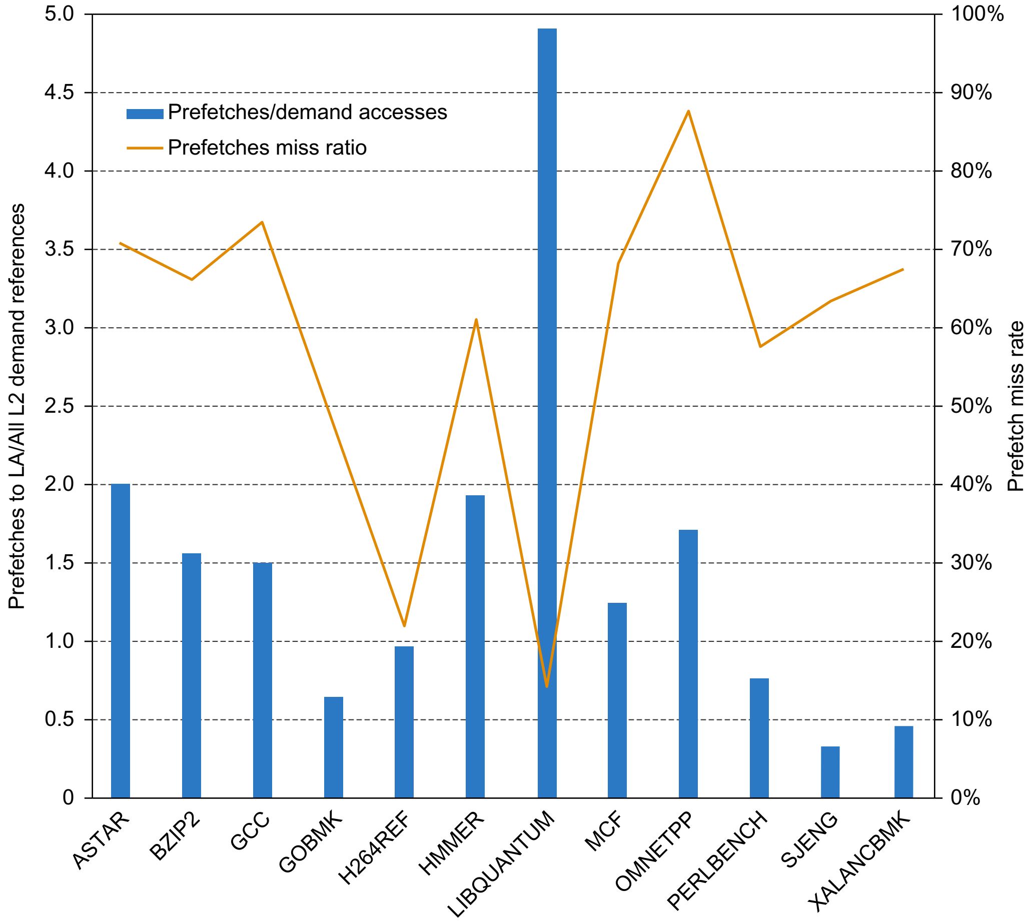


Figure 2.27 The fraction of L2 requests that are prefetches is shown via the columns and the left axis. The right axis and the line shows the prefetch hit rate. These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University, based on earlier studies of the Intel Core Duo and other processors (see Peng et al., 2008).

Because the cost for a miss to memory is over 100 cycles and the average data miss rate in L2 combining both prefetch and demand misses is over 7%, L3 is obviously critical. Without L3 and assuming that about one-third of the instructions are loads or stores, L2 cache misses could add over two cycles per instruction to the CPI! Obviously, prefetching past L2 would make no sense without an L3.

In comparison, the average L3 data miss rate of 0.5% is still significant but less than one-third of the L2 demand miss rate and 10 times less than the L1 demand miss rate. Only in two benchmarks (OMNETPP and MCF) is the L3 miss rate

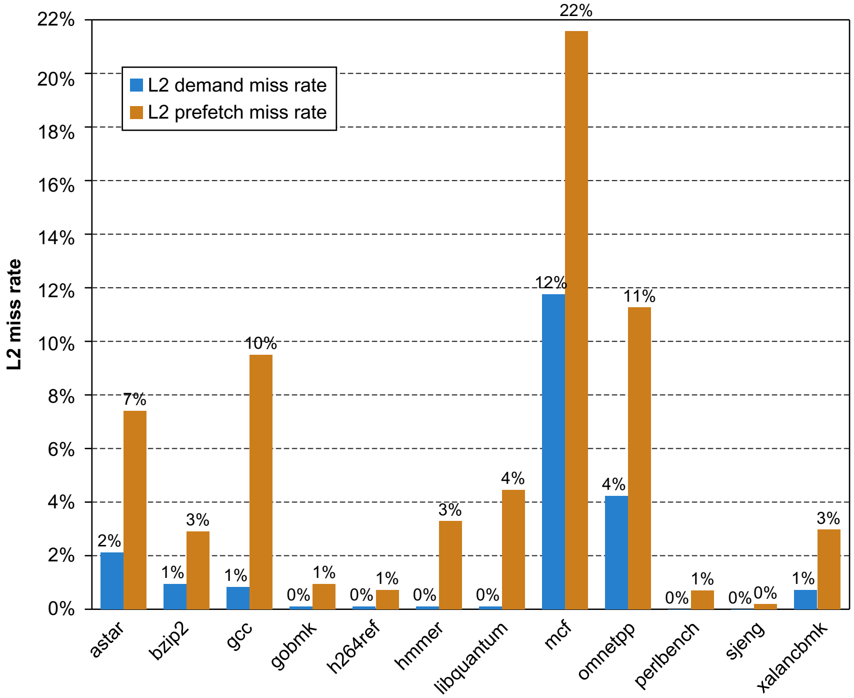


Figure 2.28 The L2 demand miss rate and prefetch miss rate, both shown relative to all the references to L1, which also includes prefetches, speculative loads that do not complete, and program-generated loads and stores (demand references). These data, like the rest in this section, were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

above 0.5%; in those two cases, the miss rate of about 2.3% likely dominates all other performance losses. In the next chapter, we will examine the relationship between the i7 CPI and cache misses, as well as other pipeline effects.

2.7 Fallacies and Pitfalls

As the most naturally quantitative of the computer architecture disciplines, memory hierarchy would seem to be less vulnerable to fallacies and pitfalls. Yet we were limited here not by lack of warnings, but by lack of space!

Fallacy *Predicting cache performance of one program from another.*

Figure 2.29 shows the instruction miss rates and data miss rates for three programs from the SPEC2000 benchmark suite as cache size varies. Depending on the

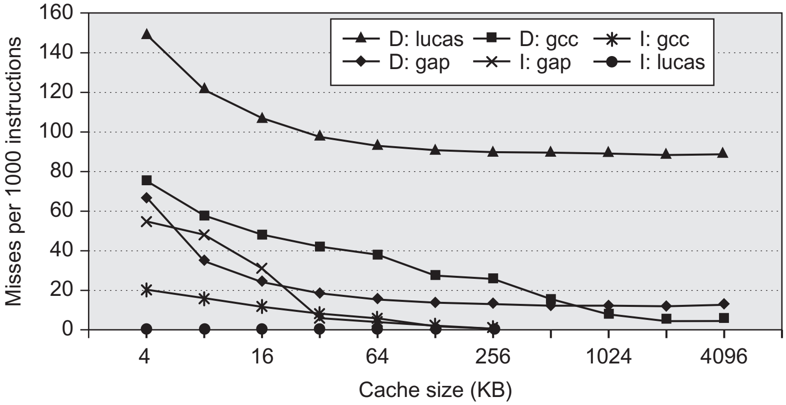


Figure 2.29 Instruction and data misses per 1000 instructions as cache size varies from 4 KiB to 4096 KiB. Instruction misses for gcc are 30,000–40,000 times larger than for lucas, and, conversely, data misses for lucas are 2–60 times larger than for gcc. The programs gap, gcc, and lucas are from the SPEC2000 benchmark suite.

program, the data misses per thousand instructions for a 4096 KiB cache are 9, 2, or 90, and the instruction misses per thousand instructions for a 4 KiB cache are 55, 19, or 0.0004. Commercial programs such as databases will have significant miss rates even in large second-level caches, which is generally not the case for the SPECCPU programs. Clearly, generalizing cache performance from one program to another is unwise. As Figure 2.24 reminds us, there is a great deal of variation, and even predictions about the relative miss rates of integer and floating-point-intensive programs can be wrong, as mcf and sphnix3 remind us!

Pitfall *Simulating enough instructions to get accurate performance measures of the memory hierarchy.*

There are really three pitfalls here. One is trying to predict performance of a large cache using a small trace. Another is that a program's locality behavior is not constant over the run of the entire program. The third is that a program's locality behavior may vary depending on the input.

Figure 2.30 shows the cumulative average instruction misses per thousand instructions for five inputs to a single SPEC2000 program. For these inputs, the average memory rate for the first 1.9 billion instructions is very different from the average miss rate for the rest of the execution.

Pitfall *Not delivering high memory bandwidth in a cache-based system.*

Caches help with average cache memory latency but may not deliver high memory bandwidth to an application that must go to main memory. The architect must design a high bandwidth memory behind the cache for such applications. We will revisit this pitfall in Chapters 4 and 5.

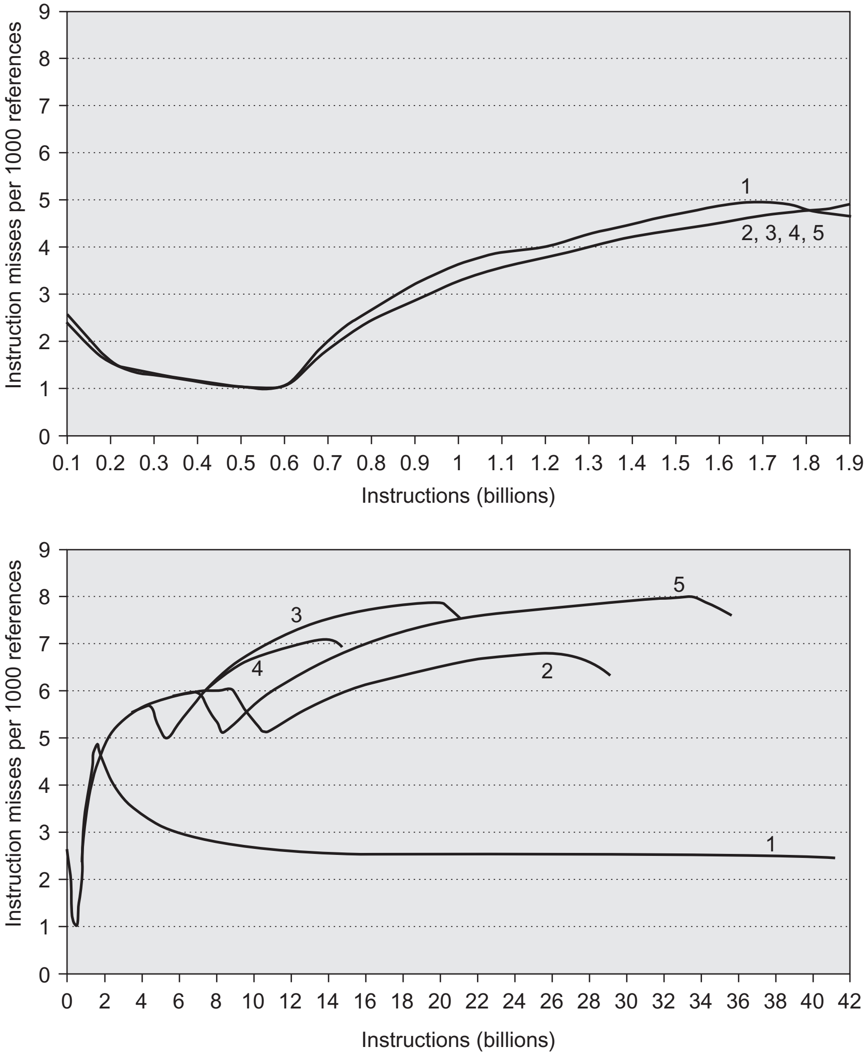


Figure 2.30 Instruction misses per 1000 references for five inputs to the perl benchmark in SPEC2000. There is little variation in misses and little difference between the five inputs for the first 1.9 billion instructions. Running to completion shows how misses vary over the life of the program and how they depend on the input. The top graph shows the running average misses for the first 1.9 billion instructions, which starts at about 2.5 and ends at about 4.7 misses per 1000 references for all five inputs. The bottom graph shows the running average misses to run to completion, which takes 16–41 billion instructions depending on the input. After the first 1.9 billion instructions, the misses per 1000 references vary from 2.4 to 7.9 depending on the input. The simulations were for the Alpha processor using separate L1 caches for instructions and data, each being two-way 64 KiB with LRU, and a unified 1 MiB direct-mapped L2 cache.

Pitfall *Implementing a virtual machine monitor on an instruction set architecture that wasn't designed to be virtualizable.*

Many architects in the 1970s and 1980s weren't careful to make sure that all instructions reading or writing information related to hardware resource information were privileged. This *laissez faire* attitude causes problems for VMMs for all of these architectures, including the 80x86, which we use here as an example.

Figure 2.31 describes the 18 instructions that cause problems for paravirtualization (Robin and Irvine, 2000). The two broad classes are instructions that

- read control registers in user mode that reveal that the guest operating system is running in a virtual machine (such as `POPF` mentioned earlier) and
- check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level.

Virtual memory is also challenging. Because the 80x86 TLBs do not support process ID tags, as do most RISC architectures, it is more expensive for the VMM and guest OSes to share the TLB; each address space change typically requires a TLB flush.

Problem category	Problem 80x86 instructions
Access sensitive registers without trapping when running in user mode	Store global descriptor table register (`SGDT`) Store local descriptor table register (`SLDT`) Store interrupt descriptor table register (`SIDT`) Store machine status word (`SMSW`) Push flags (`PUSHF`, `PUSHFD`) Pop flags (`POPF`, `POPFD`)
When accessing virtual memory mechanisms in user mode, instructions fail the 80x86 protection checks	Load access rights from segment descriptor (`LAR`) Load segment limit from segment descriptor (`LSL`) Verify if segment descriptor is readable (`VERR`) Verify if segment descriptor is writable (`VERW`) Pop to segment register (`POP CS`, `POP SS`, …) Push segment register (`PUSH CS`, `PUSH SS`, …) Far call to different privilege level (`CALL`) Far return to different privilege level (`RET`) Far jump to different privilege level (`JMP`) Software interrupt (`INT`) Store segment selector register (`STR`) Move to/from segment registers (`MOVE`)

Figure 2.31 Summary of 18 80x86 instructions that cause problems for virtualization (Robin and Irvine, 2000). The first five instructions of the top group allow a program in user mode to read a control register, such as a descriptor table register without causing a trap. The pop flags instruction modifies a control register with sensitive information but fails silently when in user mode. The protection checking of the segmented architecture of the 80x86 is the downfall of the bottom group because each of these instructions checks the privilege level implicitly as part of instruction execution when reading a control register. The checking assumes that the OS must be at the highest privilege level, which is not the case for guest VMs. Only the MOVE to segment register tries to modify control state, and protection checking foils it as well.

Virtualizing I/O is also a challenge for the 80x86, in part because it supports memory-mapped I/O and has separate I/O instructions, but more importantly because there are a very large number and variety of types of devices and device drivers of PCs for the VMM to handle. Third-party vendors supply their own drivers, and they may not properly virtualize. One solution for conventional VM implementations is to load real device drivers directly into the VMM.

To simplify implementations of VMMs on the 80x86, both AMD and Intel have proposed extensions to the architecture. Intel's VT-x provides a new execution mode for running VMs, a architected definition of the VM state, instructions to swap VMs rapidly, and a large set of parameters to select the circumstances where a VMM must be invoked. Altogether, VT-x adds 11 new instructions for the 80x86. AMD's Secure Virtual Machine (SVM) provides similar functionality.

After turning on the mode that enables VT-x support (via the new `VMXON` instruction), VT-x offers four privilege levels for the guest OS that are lower in priority than the original four (and fix issues like the problem with the `POPF` instruction mentioned earlier). VT-x captures all the states of a virtual machine in the Virtual Machine Control State (VMCS) and then provides atomic instructions to save and restore a VMCS. In addition to critical state, the VMCS includes configuration information to determine when to invoke the VMM and then specifically what caused the VMM to be invoked. To reduce the number of times the VMM must be invoked, this mode adds shadow versions of some sensitive registers and adds masks that check to see whether critical bits of a sensitive register will be changed before trapping. To reduce the cost of virtualizing virtual memory, AMD's SVM adds an additional level of indirection, called *nested page tables*, which makes shadow page tables unnecessary (see Section L.7 of Appendix L).

2.8 Concluding Remarks: Looking Ahead

> Over the past thirty years there have been several predictions of the eminent [sic] cessation of the rate of improvement in computer performance. Every such prediction was wrong. They were wrong because they hinged on unstated assumptions that were overturned by subsequent events. So, for example, the failure to foresee the move from discrete components to integrated circuits led to a prediction that the speed of light would limit computer speeds to several orders of magnitude slower than they are now. Our prediction of the memory wall is probably wrong too but it suggests that we have to start thinking "out of the box."
>
> **Wm. A. Wulf and Sally A. McKee,**
> *Hitting the Memory Wall: Implications of the Obvious,*
> Department of Computer Science, University of Virginia (December 1994).
> This paper introduced the term *memory wall.*

The possibility of using a memory hierarchy dates back to the earliest days of general-purpose digital computers in the late 1940s and early 1950s. Virtual memory was introduced in research computers in the early 1960s and into IBM mainframes in the 1970s. Caches appeared around the same time. The basic concepts have been expanded and enhanced over time to help close the access time gap between main memory and processors, but the basic concepts remain.

One trend that is causing a significant change in the design of memory hierarchies is a continued slowdown in both density and access time of DRAMs. In the past 15 years, both these trends have been observed and have been even more obvious over the past 5 years. While some increases in DRAM bandwidth have been achieved, decreases in access time have come much more slowly and almost vanished between DDR4 and DDR3. The end of Dennard scaling as well as a slowdown in Moore's Law both contributed to this situation. The trenched capacitor design used in DRAMs is also limiting its ability to scale. It may well be the case that packaging technologies such as stacked memory will be the dominant source of improvements in DRAM access bandwidth and latency.

Independently of improvements in DRAM, Flash memory has been playing a much larger role. In PMDs, Flash has dominated for 15 years and became the standard for laptops almost 10 years ago. In the past few years, many desktops have shipped with Flash as the primary secondary storage. Flash's potential advantage over DRAMs, specifically the absence of a per-bit transistor to control writing, is also its Achilles heel. Flash must use bulk erase-rewrite cycles that are considerably slower. As a result, although Flash has become the fastest growing form of secondary storage, SDRAMs still dominate for main memory.

Although phase-change materials as a basis for memory have been around for a while, they have never been serious competitors either for magnetic disks or for Flash. The recent announcement by Intel and Micron of the cross-point technology may change this. The technology appears to have several advantages over Flash, including the elimination of the slow erase-to-write cycle and greater longevity in terms. It could be that this technology will finally be the technology that replaces the electromechanical disks that have dominated bulk storage for more than 50 years!

For some years, a variety of predictions have been made about the coming memory wall (see previously cited quote and paper), which would lead to serious limits on processor performance. Fortunately, the extension of caches to multiple levels (from 2 to 4), more sophisticated refill and prefetch schemes, greater compiler and programmer awareness of the importance of locality, and tremendous improvements in DRAM bandwidth (a factor of over 150 times since the mid-1990s) have helped keep the memory wall at bay. In recent years, the combination of access time constraints on the size of L1 (which is limited by the clock cycle) and energy-related limitations on the size of L2 and L3 have raised new challenges. The evolution of the i7 processor class over 6–7 years illustrates this: the caches are the same size in the i7 6700 as they were in the first generation i7 processors! The more aggressive use of prefetching is an attempt to overcome the inability to increase L2 and L3. Off-chip L4 caches are likely to become more important because they are less energy-constrained than on-chip caches.

In addition to schemes relying on multilevel caches, the introduction of out-of-order pipelines with multiple outstanding misses has allowed available instruction-level parallelism to hide the memory latency remaining in a cache-based system. The introduction of multithreading and more thread-level parallelism takes this a step further by providing more parallelism and thus more latency-hiding opportunities. It is likely that the use of instruction- and thread-level parallelism will be a more important tool in hiding whatever memory delays are encountered in modern multilevel cache systems.

One idea that periodically arises is the use of programmer-controlled scratchpad or other high-speed visible memories, which we will see are used in GPUs. Such ideas have never made the mainstream in general-purpose processors for several reasons: First, they break the memory model by introducing address spaces with different behavior. Second, unlike compiler-based or programmer-based cache optimizations (such as prefetching), memory transformations with scratchpads must completely handle the remapping from main memory address space to the scratchpad address space. This makes such transformations more difficult and limited in applicability. In GPUs (see Chapter 4), where local scratchpad memories are heavily used, the burden for managing them currently falls on the programmer. For domain-specific software systems that can use such memories, the performance gains are very significant. It is likely that HBM technologies will thus be used for caching in large, general-purpose computers and quite possibility as the main working memories in graphics and similar systems. As domain-specific architectures become more important in overcoming the limitations arising from the end of Dennard's Law and the slowdown in Moore's Law (see Chapter 7), scratchpad memories and vector-like register sets are likely to see more use.

The implications of the end of Dennard's Law affect both DRAM and processor technology. Thus, rather than a widening gulf between processors and main memory, we are likely to see a slowdown in both technologies, leading to slower overall growth rates in performance. New innovations in computer architecture and in related software that together increase performance and efficiency will be key to continuing the performance improvements seen over the past 50 years.

2.9 Historical Perspectives and References

In Section M.3 (available online) we examine the history of caches, virtual memory, and virtual machines. IBM plays a prominent role in the history of all three. References for further reading are included.

Case Studies and Exercises by Norman P. Jouppi, Rajeev Balasubramonian, Naveen Muralimanohar, and Sheng Li

Case Study 1: Optimizing Cache Performance via Advanced Techniques

Concepts illustrated by this case study

- Nonblocking Caches
- Compiler Optimizations for Caches
- Software and Hardware Prefetching
- Calculating Impact of Cache Performance on More Complex Processors

The transpose of a matrix interchanges its rows and columns; this concept is illustrated here:

$$\begin{bmatrix} \mathtt{A11} & \mathtt{A12} & \mathtt{A13} & \mathtt{A14} \\ \mathtt{A21} & \mathtt{A22} & \mathtt{A23} & \mathtt{A24} \\ \mathtt{A31} & \mathtt{A32} & \mathtt{A33} & \mathtt{A34} \\ \mathtt{A41} & \mathtt{A42} & \mathtt{A43} & \mathtt{A44} \end{bmatrix} \Rightarrow \begin{bmatrix} \mathtt{A11} & \mathtt{A21} & \mathtt{A31} & \mathtt{A41} \\ \mathtt{A12} & \mathtt{A22} & \mathtt{A32} & \mathtt{A42} \\ \mathtt{A13} & \mathtt{A23} & \mathtt{A33} & \mathtt{A43} \\ \mathtt{A14} & \mathtt{A24} & \mathtt{A34} & \mathtt{A44} \end{bmatrix}$$

Here is a simple C loop to show the transpose:

```
for (i = 0; i < 3; i++) {
 for (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

Assume that both the input and output matrices are stored in the row major order (*row major order* means that the row index changes fastest). Assume that you are executing a 256·256 double-precision transpose on a processor with a 16 KB fully associative (don't worry about cache conflicts) least recently used (LRU) replacement L1 data cache with 64-byte blocks. Assume that the L1 cache misses or prefetches require 16 cycles and always hit in the L2 cache, and that the L2 cache can process a request every 2 processor cycles. Assume that each iteration of the preceding inner loop requires 4 cycles if the data are present in the L1 cache. Assume that the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically, assume that writing back dirty cache blocks requires 0 cycles.

2.1 [10/15/15/12/20] <2.3> For the preceding simple implementation, this execution order would be nonideal for the input matrix; however, applying a loop interchange optimization would create a nonideal order for the output matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <2.3> What should be the minimum size of the cache to take advantage of blocked execution?

b. [15] <2.3> How do the relative number of misses in the blocked and unblocked versions compare in the preceding minimum-sized cache?

c. [15] <2.3> Write code to perform a transpose with a block size parameter B that uses $B \cdot B$ blocks.

d. [12] <2.3> What is the minimum associativity required of the L1 cache for consistent performance independent of both arrays' position in memory?

e. [20] <2.3> Try out blocked and nonblocked 256·256 matrix transpositions on a computer. How closely do the results match your expectations based on what you know about the computer's memory system? Explain any discrepancies if possible.

2.2 [10] <2.3> Assume you are designing a hardware prefetcher for the preceding *unblocked* matrix transposition code. The simplest type of hardware prefetcher only prefetches sequential cache blocks after a miss. More complicated "nonunit stride" hardware prefetchers can analyze a miss reference stream and detect and prefetch nonunit strides. In contrast, software prefetching can determine nonunit strides as easily as it can determine unit strides. Assume prefetches write directly into the cache and that there is no "pollution" (overwriting data that must be used before the data that are prefetched). For best performance given a nonunit stride prefetcher, in the steady state of the inner loop, how many prefetches must be outstanding at a given time?

2.3 [15/20] <2.3> With software prefetching, it is important to be careful to have the prefetches occur in time for use but also to minimize the number of outstanding prefetches to live within the capabilities of the microarchitecture and minimize cache pollution. This is complicated by the fact that different processors have different capabilities and limitations.

a. [15] <2.3> Create a blocked version of the matrix transpose with software prefetching.

b. [20] <2.3> Estimate and compare the performance of the blocked and unblocked transpose codes both with and without software prefetching.

Case Study 2: Putting It All Together: Highly Parallel Memory Systems

Concept illustrated by this case study

- Cross-Cutting Issues: The Design of Memory Hierarchies

The program in Figure 2.32 can be used to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Figure 2.32 shows the code in C. The first part is a procedure that uses a standard utility to get an accurate measure of the user CPU time; this procedure may have to be changed to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The results are output in `.csv` file format to facilitate importing into spreadsheets. You may need to change `CACHE_MAX` depending on the question you are answering and the size of memory on the system you are measuring. Running the program in single-user mode or at least without other active applications will give more consistent results. The code in Figure 2.32 was derived from a program written by Andrea Dusseau at the University of California-Berkeley and was based on a detailed description found in Saavedra-Barrera (1992). It has been modified to fix a number of issues with more modern machines and to run under Microsoft

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
   __time64_t ltime;
   _time64( &ltime );
   return (double) ltime;
}
int label(int i) {/* generate text labels */
   if (i<1e3) printf("%1dB,",i);
   else if (i<1e6) printf("%1dK,",i/1024);
   else if (i<1e9) printf("%1dM,",i/1048576);
   else printf("%1dG,",i/1073741824);
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
   label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
   label(csize*sizeof(int)); /* print cache size this loop */
   for (stride=1; stride <= csize/2; stride=stride*2) {

      /* Lay out path of memory references in array */
      for (index=0; index < csize; index=index+stride)
         x[index] = index + stride; /* pointer to next */
      x[index-stride] = 0; /* loop back to beginning */

      /* Wait for timer to roll over */
      lastsec = get_seconds();
       sec0 = get_seconds(); while (sec0 == lastsec);

      /* Walk through path in array for twenty seconds */
      /* This gives 5% accuracy with second resolution */
      steps = 0.0; /* number of steps taken */
      nextstep = 0; /* start at beginning of path */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until collect 20 seconds */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               nextstep = 0;
               do nextstep = x[nextstep]; /* dependency */
               while (nextstep != 0);
         }
         steps = steps + 1.0; /* count loop iterations */
         sec1 = get_seconds(); /* end timer */
      } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
      sec = sec1 - sec0;

      /* Repeat empty loop to loop subtract overhead */
      tsteps = 0.0; /* used to match no. while iterations */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until same no. iterations as above */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               index = 0;
               do index = index + stride;
               while (index < csize);
         }
         tsteps = tsteps + 1.0;
         sec1 = get_seconds(); /* - overhead */
      } while (tsteps<steps); /* until = no. iterations */
      sec = sec - (sec1 - sec0);
      loadtime = (sec*1e9)/(steps*csize);
      /* write out results in .csv format for Excel */
      printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
     }; /* end of inner for loop */
     printf("\n");
   }; /* end of outer for loop */
   return 0;
}
```

Figure 2.32 C program for evaluating memory system.

Visual C++. It can be downloaded from http://www.hpl.hp.com/research/cacti/aca_ch2_cs2.c.

The preceding program assumes that program addresses track physical addresses, which is true on the few machines that use virtually addressed caches, such as the Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine in order to get smooth lines in your results. To answer the following questions, assume that the sizes of all components of the memory hierarchy are powers of 2. Assume that the size of the page is much larger than the size of a block in a second-level cache (if there is one) and that the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache. An example of the output of the program is plotted in Figure 2.33; the key lists the size of the array that is exercised.

2.4 [12/12/12/10/12] <2.6> Using the sample program results in Figure 2.33:

a. [12] <2.6> What are the overall size and block size of the second-level cache?

b. [12] <2.6> What is the miss penalty of the second-level cache?

c. [12] <2.6> What is the associativity of the second-level cache?

d. [10] <2.6> What is the size of the main memory?

e. [12] <2.6> What is the paging time if the page size is 4 KB?

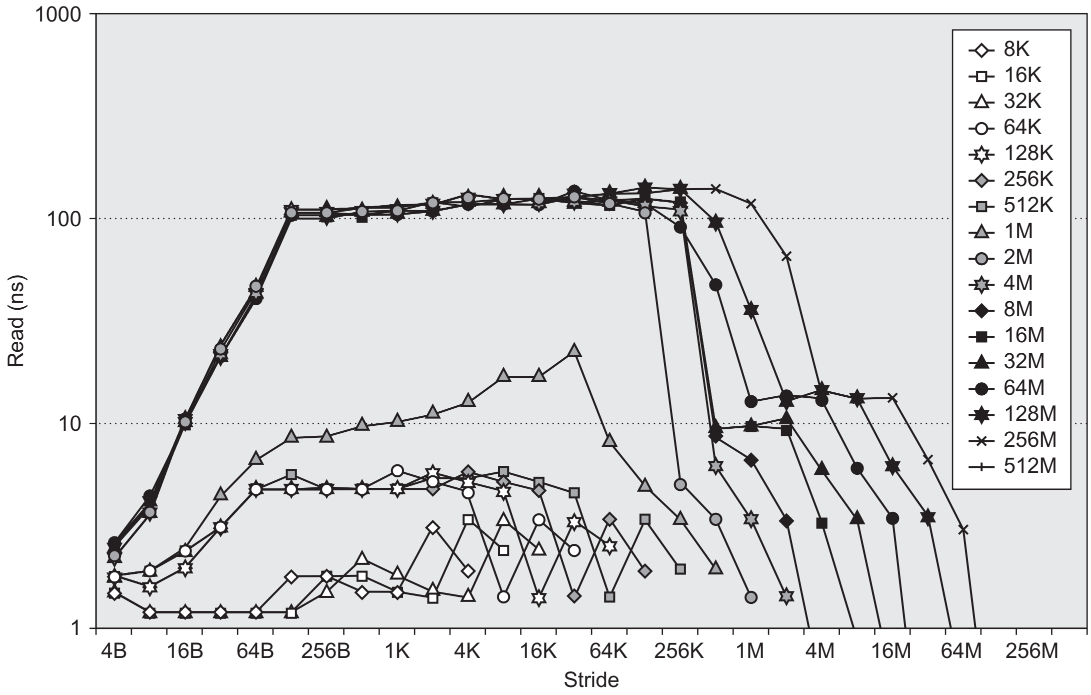


Figure 2.33 Sample results from program in Figure 2.32.

2.5 [12/15/15/20] <2.6> If necessary, modify the code in Figure 2.32 to measure the following system characteristics. Plot the experimental results with elapsed time on the *y*-axis and the memory stride on the *x*-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

a. [12] <2.6> What is the system page size?

b. [15] <2.6> How many entries are there in the TLB?

c. [15] <2.6> What is the miss penalty for the TLB?

d. [20] <2.6> What is the associativity of the TLB?

2.6 [20/20] <2.6> In multiprocessor memory systems, lower levels of the memory hierarchy may not be able to be saturated by a single processor but should be able to be saturated by multiple processors working together. Modify the code in Figure 2.32, and run multiple copies at the same time. Can you determine:

a. [20] <2.6> How many actual processors are in your computer system and how many system processors are just additional multithreaded contexts?

b. [20] <2.6> How many memory controllers does your system have?

2.7 [20] <2.6> Can you think of a way to test some of the characteristics of an instruction cache using a program? *Hint*: The compiler may generate a large number of nonobvious instructions from a piece of code. Try to use simple arithmetic instructions of known length in your instruction set architecture (ISA).

Case Study 3: Studying the Impact of Various Memory System Organizations

Concepts illustrated by this case study

- DDR3 memory systems
- Impact of ranks, banks, row buffers on performance and power
- DRAM timing parameters

A processor chip typically supports a few DDR3 or DDR4 memory channels. We will focus on a single memory channel in this case study and explore how its performance and power are impacted by varying several parameters. Recall that the channel is populated with one or more DIMMs. Each DIMM supports one or more ranks—a rank is a collection of DRAM chips that work in unison to service a single command issued by the memory controller. For example, a rank may be composed of 16 DRAM chips, where each chip deals with a 4-bit input or output on every channel clock edge. Each such chip is referred to as a ×4 (by four) chip. In other examples, a rank may be composed of 8 × 8 chips or 4 × 16 chips—note that in each case, a rank can handle data that are being placed on a 64-bit memory channel. A rank is itself partitioned into 8 (DDR3) or 16 (DDR4) banks. Each bank has a row buffer that essentially remembers the last row read out of a bank. Here's an example of a typical sequence of memory commands when performing a read from a bank:

(i) The memory controller issues a Precharge command to get the bank ready to access a new row. The precharge is completed after time tRP.

(ii) The memory controller then issues an Activate command to read the appropriate row out of the bank. The activation is completed after time tRCD and the row is deemed to be part of the row buffer.

(iii) The memory controller can then issue a column-read or CAS command that places a specific subset of the row buffer on the memory channel. After time CL, the first 64 bits of the data burst are placed on the memory channel. A burst typically includes eight 64-bit transfers on the memory channel, performed on the rising and falling edges of 4 memory clock cycles (referred to as transfer time).

(iv) If the memory controller wants to then access data in a different row of the bank, referred to as a row buffer miss, it repeats steps (i)–(iii). For now, we will assume that after CL has elapsed, the Precharge in step (i) can be issued; in some cases, an additional delay must be added, but we will ignore that delay here. If the memory controller wants to access another block of data in the same row, referred to as a row buffer hit, it simply issues another CAS command. Two back-to-back CAS commands have to be separated by at least 4 cycles so that the first data transfer is complete before the second data transfer can begin.

Note that a memory controller can issue commands to different banks in successive cycles so that it can perform many memory reads/writes in parallel and it is not sitting idle waiting for tRP, tRCD, and CL to elapse in a single bank. For the subsequent questions, assume that $tRP = tRCD = CL = 13$ ns, and that the memory channel frequency is 1 GHz, that is, a transfer time of 4 ns.

2.8 [10] <2.2> What is the read latency experienced by a memory controller on a row buffer miss?

2.9 [10] <2.2> What is the latency experienced by a memory controller on a row buffer hit?

2.10 [10] <2.2> If the memory channel supports only one bank and the memory access pattern is dominated by row buffer misses, what is the utilization of the memory channel?

2.11 [15] <2.2> Assuming a 100% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve a 100% memory channel utilization?

2.12 [10] <2.2> Assuming a 50% row buffer miss rate, what is the minimum number of banks that the memory channel should support in order to achieve a 100% memory channel utilization?

2.13 [15] <2.2> Assume that we are executing an application with four threads and the threads exhibit zero spatial locality, that is, a 100% row buffer miss rate. Every 200 ns, each of the four threads simultaneously inserts a read operation into the memory controller queue. What is the average memory latency experienced if the memory channel supports only one bank? What if the memory channel supported four banks?

2.14 [10] <2.2> From these questions, what have you learned about the benefits and downsides of growing the number of banks?

2.15 [20] <2.2> Now let's turn our attention to memory power. Download a copy of the Micron power calculator from this link: https://www.micron.com/~/media/documents/products/power-calculator/ddr3_power_calc.xlsm. This spreadsheet is preconfigured to estimate the power dissipation in a single 2 Gb ×8 DDR3 SDRAM memory chip manufactured by Micron. Click on the "Summary" tab to see the power breakdown in a single DRAM chip under default usage conditions (reads occupy the channel for 45% of all cycles, writes occupy the channel for 25% of all cycles, and the row buffer hit rate is 50%). This chip consumes 535 mW, and the breakdown shows that about half of that power is expended in Activate operations, about 38% in CAS operations, and 12% in background power. Next, click on the "System Config" tab. Modify the read/write traffic and the row buffer hit rate and observe how that changes the power profile. For example, what is the decrease in power when channel utilization is 35% (25% reads and 10% writes), or when row buffer hit rate is increased to 80%?

2.16 [20] <2.2> In the default configuration, a rank consists of eight ×8 2 Gb DRAM chips. A rank can also comprise16 × 4 chips or 4 × 16 chips. You can also vary the capacity of each DRAM chip—1 Gb, 2 Gb, and 4 Gb. These selections can be made in the "DDR3 Config" tab of the Micron power calculator. Tabulate the total power consumed for each rank organization. What is the most power-efficient approach to constructing a rank of a given capacity?

Exercises

2.17 [12/12/15] <2.3> The following questions investigate the impact of small and simple caches using CACTI and assume a 65 nm (0.065 m) technology. (CACTI is available in an online form at http://quid.hpl.hp.com:9081/cacti/.)

a. [12] <2.3> Compare the access times of 64 KB caches with 64-byte blocks and a single bank. What are the relative access times of two-way and four-way set associative caches compared to a direct mapped organization?

b. [12] <2.3> Compare the access times of four-way set associative caches with 64-byte blocks and a single bank. What are the relative access times of 32 and 64 KB caches compared to a 16 KB cache?

c. [15] <2.3> For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time given that misses per instruction for a certain workload suite is 0.00664 for direct-mapped, 0.00366 for two-way set associative, 0.000987 for four-way set associative, and 0.000266 for eight-way set associative cache. Overall, there are 0.3 data references per instruction. Assume cache misses take 10 ns in all models. To calculate the hit time in cycles, assume the cycle time output using CACTI, which corresponds to the maximum frequency a cache can operate without any bubbles in the pipeline.

2.18 [12/15/15/10] <2.3> You are investigating the possible benefits of a way-predicting L1 cache. Assume that a 64 KB four-way set associative single-banked L1 data cache is the cycle time limiter in a system. For an alternative cache organization, you are considering a way-predicted cache modeled as a 64 KB direct-mapped cache with 80% prediction accuracy. Unless stated otherwise, assume that a mispredicted way access that hits in the cache takes one more cycle. Assume the miss rates and the miss penalties in question 2.8 part (c).

a. [12] <2.3> What is the average memory access time of the current cache (in cycles) versus the way-predicted cache?

b. [15] <2.3> If all other components could operate with the faster way-predicted cache cycle time (including the main memory), what would be the impact on performance from using the way-predicted cache?

c. [15] <2.3> Way-predicted caches have usually been used only for instruction caches that feed an instruction queue or buffer. Imagine that you want to try out way prediction on a data cache. Assume that you have 80% prediction accuracy and that subsequent operations (e.g., data cache access of other instructions, dependent operations) are issued assuming a correct way prediction. Thus a way misprediction necessitates a pipe flush and replay trap, which requires 15 cycles. Is the change in average memory access time per load instruction with data cache way prediction positive or negative, and how much is it?

d. [10] <2.3> As an alternative to way prediction, many large associative L2 caches serialize tag and data access so that only the required dataset array needs to be activated. This saves power but increases the access time. Use CACTI's detailed web interface for a 0.065 m process 1 MB four-way set associative cache with 64-byte blocks, 144 bits read out, 1 bank, only 1 read/write port, 30 bit tags, and ITRS-HP technology with global wires. What is the ratio of the access times for serializing tag and data access compared to parallel access?

2.19 [10/12] <2.3> You have been asked to investigate the relative performance of a banked versus pipelined L1 data cache for a new microprocessor. Assume a 64 KB two-way set associative cache with 64-byte blocks. The pipelined cache would consist of three pipe stages, similar in capacity to the Alpha 21264 data cache. A banked implementation would consist of two 32 KB two-way set associative banks. Use CACTI and assume a 65 nm (0.065 m) technology to answer the following questions. The cycle time output in the web version shows at what frequency a cache can operate without any bubbles in the pipeline.

a. [10] <2.3> What is the cycle time of the cache in comparison to its access time, and how many pipe stages will the cache take up (to two decimal places)?

b. [12] <2.3> Compare the area and total dynamic read energy per access of the pipelined design versus the banked design. State which takes up less area and which requires more power, and explain why that might be.

2.20 [12/15] <2.3> Consider the usage of critical word first and early restart on L2 cache misses. Assume a 1 MB L2 cache with 64-byte blocks and a refill path that is 16 bytes wide. Assume that the L2 can be written with 16 bytes every 4 processor cycles, the time to receive the first 16 byte block from the memory controller is 120 cycles, each additional 16 byte block from main memory requires 16 cycles, and data can be bypassed directly into the read port of the L2 cache. Ignore any cycles to transfer the miss request to the L2 cache and the requested data to the L1 cache.

a. [12] <2.3> How many cycles would it take to service an L2 cache miss with and without critical word first and early restart?

b. [15] <2.3> Do you think critical word first and early restart would be more important for L1 caches or L2 caches, and what factors would contribute to their relative importance?

2.21 [12/12] <2.3> You are designing a write buffer between a write-through L1 cache and a write-back L2 cache. The L2 cache write data bus is 16 B wide and can perform a write to an independent cache address every four processor cycles.

a. [12] <2.3> How many bytes wide should each write buffer entry be?

b. [15] <2.3> What speedup could be expected in the steady state by using a merging write buffer instead of a nonmerging buffer when zeroing memory by the execution of 64-bit stores if all other instructions could be issued in parallel with the stores and the blocks are present in the L2 cache?

c. [15] <2.3> What would the effect of possible L1 misses be on the number of required write buffer entries for systems with blocking and nonblocking caches?

2.22 [20] <2.1, 2.2, 2.3> A cache acts as a filter. For example, for every 1000 instructions of a program, an average of 20 memory accesses may exhibit low enough locality that they cannot be serviced by a 2 MB cache. The 2 MB cache is said to have an MPKI (misses per thousand instructions) of 20, and this will be largely true regardless of the smaller caches that precede the 2 MB cache. Assume the following cache/latency/MPKI values: 32 KB/1/100, 128 KB/2/80, 512 KB/4/50, 2 MB/8/40, 8 MB/16/10. Assume that accessing the off-chip memory system requires 200 cycles on average. For the following cache configurations, calculate the average time spent accessing the cache hierarchy. What do you observe about the downsides of a cache hierarchy that is too shallow or too deep?

a. 32 KB L1; 8 MB L2; off-chip memory

b. 32 KB L1; 512 KB L2; 8 MB L3; off-chip memory

c. 32 KB L1; 128 KB L2; 2 MB L3; 8 MB L4; off-chip memory

2.23 [15] <2.1, 2.2, 2.3> Consider a 16 MB 16-way L3 cache that is shared by two programs A and B. There is a mechanism in the cache that monitors cache miss rates for each program and allocates 1–15 ways to each program such that the overall number of cache misses is reduced. Assume that program A has an MPKI of 100 when it is assigned 1 MB of the cache. Each additional 1 MB assigned to program

A reduces the MPKI by 1. Program B has an MPKI of 50 when it is assigned 1 MB of cache; each additional 1 MB assigned to program B reduces its MPKI by 2. What is the best allocation of ways to programs A and B?

2.24 [20] <2.1, 2.6> You are designing a PMD and optimizing it for low energy. The core, including an 8 KB L1 data cache, consumes 1 W whenever it is not in hibernation. If the core has a perfect L1 cache hit rate, it achieves an average CPI of 1 for a given task, that is, 1000 cycles to execute 1000 instructions. Each additional cycle accessing the L2 and beyond adds a stall cycle for the core. Based on the following specifications, what is the size of L2 cache that achieves the lowest energy for the PMD (core, L1, L2, memory) for that given task?

a. The core frequency is 1 GHz, and the L1 has an MPKI of 100.

b. A 256 KB L2 has a latency of 10 cycles, an MPKI of 20, a background power of 0.2 W, and each L2 access consumes 0.5 nJ.

c. A 1 MB L2 has a latency of 20 cycles, an MPKI of 10, a background power of 0.8 W, and each L2 access consumes 0.7 nJ.

d. The memory system has an average latency of 100 cycles, a background power of 0.5 W, and each memory access consumes 35 nJ.

2.25 [15] <2.1, 2.6> You are designing a PMD that is optimized for low power. Qualitatively explain the impact on cache hierarchy (L2 and memory) power and overall application energy if you design an L2 cache with:

a. Small block size

b. Small cache size

c. High associativity

2.30 [10/10] <2.1, 2.2, 2.3> The ways of a set can be viewed as a priority list, ordered from high priority to low priority. Every time the set is touched, the list can be reorganized to change block priorities. With this view, cache management policies can be decomposed into three sub-policies: Insertion, Promotion, and Victim Selection. Insertion defines where newly fetched blocks are placed in the priority list. Promotion defines how a block's position in the list is changed every time it is touched (a cache hit). Victim Selection defines which entry of the list is evicted to make room for a new block when there is a cache miss.

a. Can you frame the LRU cache policy in terms of the Insertion, Promotion, and Victim Selection sub-policies?

b. Can you define other Insertion and Promotion policies that may be competitive and worth exploring further?

2.31 [15] <2.1, 2.3> In a processor that is running multiple programs, the last-level cache is typically shared by all the programs. This leads to interference, where one program's behavior and cache footprint can impact the cache available to other programs. First, this is a problem from a quality-of-service (QoS) perspective, where the interference leads to a program receiving fewer resources and lower performance than promised, say by the operator of a cloud service. Second, this is a problem in terms of privacy. Based on the interference it sees, a program can infer the memory access patterns of other programs. This is referred to as a timing channel, a form of information leakage from one program to others that can be exploited to compromise data privacy or to reverse-engineer a competitor's algorithm. What policies can you add to your last-level cache so that the behavior of one program is immune to the behavior of other programs sharing the cache?

2.32 [15] <2.3> A large multimegabyte L3 cache can take tens of cycles to access because of the long wires that have to be traversed. For example, it may take 20 cycles to access a 16 MB L3 cache. Instead of organizing the 16 MB cache such that every access takes 20 cycles, we can organize the cache so that it is an array of smaller cache banks. Some of these banks may be closer to the processor core, while others may be further. This leads to nonuniform cache access (NUCA), where 2 MB of the cache may be accessible in 8 cycles, the next 2 MB in 10 cycles, and so on until the last 2 MB is accessed in 22 cycles. What new policies can you introduce to maximize performance in a NUCA cache?

2.33 [10/10/10] <2.2> Consider a desktop system with a processor connected to a 2 GB DRAM with *error-correcting code (ECC)*. Assume that there is only one memory channel of width 72 bits (64 bits for data and 8 bits for ECC).

a. [10] <2.2> How many DRAM chips are on the DIMM if 1 Gb DRAM chips are used, and how many data I/Os must each DRAM have if only one DRAM connects to each DIMM data pin?

b. [10] <2.2> What burst length is required to support 32 B L2 cache blocks?

c. [10] <2.2> Calculate the peak bandwidth for DDR2-667 and DDR2-533 DIMMs for reads from an active page excluding the ECC overhead.

2.34 [10/10] <2.2> A sample DDR2 SDRAM timing diagram is shown in Figure 2.34. tRCD is the time required to activate a row in a bank, and column address strobe (CAS) latency (CL) is the number of cycles required to read out a column in a row. Assume that the RAM is on a standard DDR2 DIMM with ECC, having 72 data lines. Also assume burst lengths of 8 that read out 8 bits, or a total of 64 B from the DIMM. Assume `tRCD = CAS (or CL) clock_frequency`, and `clock_frequency = transfers_per_second/2`. The on-chip latency

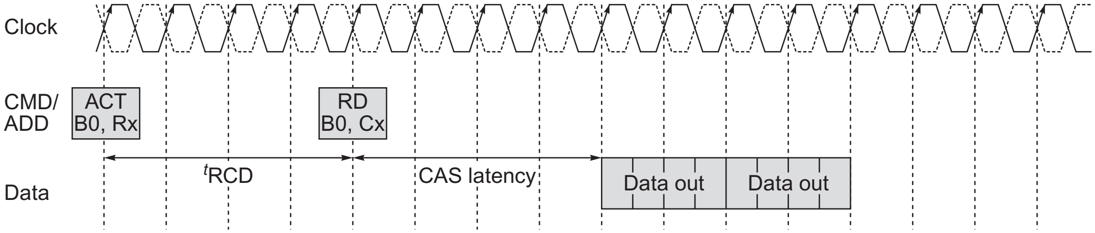


Figure 2.34 DDR2 SDRAM timing diagram.

on a cache miss through levels 1 and 2 and back, not including the DRAM access, is 20 ns.

a. [10] <2.2> How much time is required from presentation of the activate command until the last requested bit of data from the DRAM transitions from valid to invalid for the DDR2-667 1 Gb CL = 5 DIMM? Assume that for every request, we automatically prefetch another adjacent cache line in the same page.

b. [10] <2.2> What is the relative latency when using the DDR2-667 DIMM of a read requiring a bank activate versus one to an already open page, including the time required to process the miss inside the processor?

2.35 [15] <2.2> Assume that a DDR2-667 2 GB DIMM with CL = 5 is available for 130 and a DDR2-533 2 GB DIMM with CL = 4 is available for 100. Assume that two DIMMs are used in a system, and the rest of the system costs 800. Consider the performance of the system using the DDR2-667 and DDR2-533 DIMMs on a workload with 3.33 L2 misses per 1K instructions, and assume that 80% of all DRAM reads require an activate. What is the cost-performance of the entire system when using the different DIMMs, assuming only one L2 miss is outstanding at a time and an in-order core with a CPI of 1.5 not including L2 cache miss memory access time?

2.36 [12] <2.2> You are provisioning a server with eight-core 3 GHz CMP that can execute a workload with an overall CPI of 2.0 (assuming that L2 cache miss refills are not delayed). The L2 cache line size is 32 bytes. Assuming the system uses DDR2-667 DIMMs, how many independent memory channels should be provided so the system is not limited by memory bandwidth if the bandwidth required is sometimes twice the average? The workloads incur, on average, 6.67 L2 misses per 1 K instructions.

2.37 [15] <2.2> Consider a processor that has four memory channels. Should consecutive memory blocks be placed in the same bank, or should they be placed in different banks on different channels?

2.38 [12/12] <2.2> A large amount (more than a third) of DRAM power can be due to page activation (see http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf and http://www.micron.com/systemcalc). Assume you are building a system with 2 GB of memory using either 8-bank 2 Gb ×8 DDR2 DRAMs or 8-bank 1 Gb ×8 DRAMs, both with the same speed grade. Both use a page size of 1 KB, and the last-level cache line size is 64 bytes. Assume that DRAMs that are not active are in precharged standby and dissipate negligible power. Assume that the time to transition from standby to active is not significant.

a. [12] <2.2> Which type of DRAM would be expected to provide the higher system performance? Explain why.

b. [12] <2.2> How does a 2 GB DIMM made of 1 Gb ×8 DDR2 DRAMs compare with a DIMM with similar capacity made of 1 Gb ×4 DDR2 DRAMs in terms of power?

2.39 [20/15/12] <2.2> To access data from a typical DRAM, we first have to activate the appropriate row. Assume that this brings an entire page of size 8 KB to the row buffer. Then we select a particular column from the row buffer. If subsequent accesses to DRAM are to the same page, then we can skip the activation step; otherwise, we have to close the current page and precharge the bitlines for the next activation. Another popular DRAM policy is to proactively close a page and precharge bitlines as soon as an access is over. Assume that every read or write to DRAM is of size 64 bytes and DDR bus latency (data from Figure 2.33) for sending 512 bits is Tddr.

a. [20] <2.2> Assuming DDR2-667, if it takes five cycles to precharge, five cycles to activate, and four cycles to read a column, for what value of the row buffer hit rate (*r*) will you choose one policy over another to get the best access time? Assume that every access to DRAM is separated by enough time to finish a random new access.

b. [15] <2.2> If 10% of the total accesses to DRAM happen back to back or contiguously without any time gap, how will your decision change?

c. [12] <2.2> Calculate the difference in average DRAM energy per access between the two policies using the previously calculated row buffer hit rate. Assume that precharging requires 2 nJ and activation requires 4 nJ and that 100 pJ/bit are required to read or write from the row buffer.

2.40 [15] <2.2> Whenever a computer is idle, we can either put it in standby (where DRAM is still active) or we can let it hibernate. Assume that, to hibernate, we have to copy just the contents of DRAM to a nonvolatile medium such as Flash. If reading or writing a cache line of size 64 bytes to Flash requires 2.56 J and DRAM requires 0.5 nJ, and if idle power consumption for DRAM is 1.6 W (for 8 GB), how long should a system be idle to benefit from hibernating? Assume a main memory of size 8 GB.

2.41 [10/10/10/10/10] <2.4> Virtual machines (VMs) have the potential for adding many beneficial capabilities to computer systems, such as improved total cost of ownership (TCO) or availability. Could VMs be used to provide the following capabilities? If so, how could they facilitate this?

a. [10] <2.4> Test applications in production environments using development machines?

b. [10] <2.4> Quick redeployment of applications in case of disaster or failure?

c. [10] <2.4> Higher performance in I/O-intensive applications?

d. [10] <2.4> Fault isolation between different applications, resulting in higher availability for services?

e. [10] <2.4> Performing software maintenance on systems while applications are running without significant interruption?

2.42 [10/10/12/12] <2.4> Virtual machines can lose performance from a number of events, such as the execution of privileged instructions, TLB misses, traps, and I/O.

Benchmark	**Native**	**Pure**	**Para**
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install signal handler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

Figure 2.35 Early performance of various system calls under native execution, pure virtualization, and paravirtualization.

These events are usually handled in system code. Thus one way of estimating the slowdown when running under a VM is the percentage of application execution time in system versus user mode. For example, an application spending 10% of its execution in system mode might slow down by 60% when running on a VM. Figure 2.35 lists the early performance of various system calls under native execution, pure virtualization, and paravirtualization for LMbench using Xen on an Itanium system with times measured in microseconds (courtesy of Matthew Chapman of the University of New South Wales).

a. [10] <2.4> What types of programs would be expected to have smaller slowdowns when running under VMs?

b. [10] <2.4> If slowdowns were linear as a function of system time, given the preceding slowdown, how much slower would a program spending 20% of its execution in system time be expected to run?

c. [12] <2.4> What is the median slowdown of the system calls in the table above under pure virtualization and paravirtualization?

d. [12] <2.4> Which functions in the table above have the largest slowdowns? What do you think the cause of this could be?

2.43 [12] <2.4> Popek and Goldberg's definition of a virtual machine said that it would be indistinguishable from a real machine except for its performance. In this question, we will use that definition to find out if we have access to native execution on a processor or are running on a virtual machine. The Intel VT-x technology effectively provides a second set of privilege levels for the use of the virtual machine. What would a virtual machine running on top of another virtual machine have to do, assuming VT-x technology?

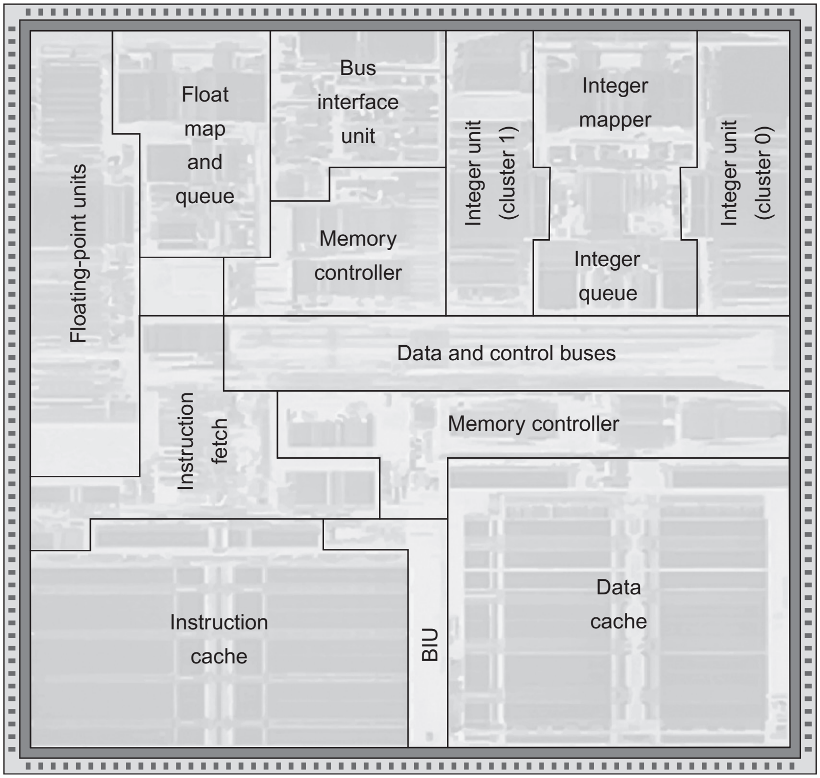
2.44 [20/25] <2.4> With the adoption of virtualization support on the x86 architecture, virtual machines are actively evolving and becoming mainstream. Compare and contrast the Intel VT-x and AMD's AMD-V virtualization technologies.

(Information on AMD-V can be found at http://sites.amd.com/us/business/it-solutions/virtualization/Pages/resources.aspx.)

a. [20] <2.4> Which one could provide higher performance for memory-intensive applications with large memory footprints?

b. [25] <2.4> Information on AMD's IOMMU support for virtualized I/O can be found at http://developer.amd.com/documentation/articles/pages/892006101.aspx. What do Virtualization Technology and an input/output memory management unit (IOMMU) do to improve virtualized I/O performance?

2.45 [30] <2.2, 2.3> Since instruction-level parallelism can also be effectively exploited on in-order superscalar processors and *very long instruction word (VLIW)* processors with speculation, one important reason for building an out-of-order (OOO) superscalar processor is the ability to tolerate unpredictable memory latency caused by cache misses. Thus you can think about hardware supporting OOO issue as being part of the memory system. Look at the floorplan of the Alpha 21264 in Figure 2.36 to find the relative area of the integer and floating-point issue queues and mappers versus the caches. The queues schedule instructions for issue,



Floating-point units
Float map and queue
Bus interface unit
Memory controller
Integer unit (cluster 1)
Integer mapper
Integer queue
Integer unit (cluster 0)
Data and control buses
Instruction fetch
Memory controller
Instruction cache
BIU
Data cache

Figure 2.36 Floorplan of the Alpha 21264 [Kessler 1999].

and the mappers rename register specifiers. Therefore these are necessary additions to support OOO issue. The 21264 only has L1 data and instruction caches on chip, and they are both 64 KB two-way set associative. Use an OOO superscalar simulator such as SimpleScalar (http://www.cs.wisc.edu/~mscalar/simplescalar.html) on memory-intensive benchmarks to find out how much performance is lost if the area of the issue queues and mappers is used for additional L1 data cache area in an in-order superscalar processor, instead of OOO issue in a model of the 21264. Make sure the other aspects of the machine are as similar as possible to make the comparison fair. Ignore any increase in access or cycle time from larger caches and effects of the larger data cache on the floorplan of the chip. (Note that this comparison will not be totally fair, as the code will not have been scheduled for the in-order processor by the compiler.)

2.46 [15] <2.2, 2.7> As discussed in Section 2.7, the Intel i7 processor has an aggressive prefetcher. What are potential disadvantages in designing a prefetcher that is extremely aggressive?

2.47 [20/20/20] <2.6> The Intel performance analyzer VTune can be used to make many measurements of cache behavior. A free evaluation version of VTune on both Windows and Linux can be downloaded from http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/. The program (`aca_ch2_cs2.c`) used in Case Study 2 has been modified so that it can work with VTune out of the box on Microsoft Visual C++. The program can be downloaded from http://www.hpl.hp.com/research/cacti/aca_ch2_cs2_vtune.c. Special VTune functions have been inserted to exclude initialization and loop overhead during the performance analysis process. Detailed VTune setup directions are given in the README section in the program. The program keeps looping for 20 seconds for every configuration. In the following experiment, you can find the effects of data size on cache and overall processor performance. Run the program in VTune on an Intel processor with the input dataset sizes of 8 KB, 128 KB, 4 MB, and 32 MB, and keep a stride of 64 bytes (stride one cache line on Intel i7 processors). Collect statistics on overall performance and L1 data cache, L2, and L3 cache performance.

a. [20] <2.6> List the number of misses per 1K instruction of L1 data cache, L2, and L3 for each dataset size and your processor model and speed. Based on the results, what can you say about the L1 data cache, L2, and L3 cache sizes on your processor? Explain your observations.

b. [20] <2.6> List the *instructions per clock* (IPC) for each dataset size and your processor model and speed. Based on the results, what can you say about the L1, L2, and L3 miss penalties on your processor? Explain your observations.

c. [20] <2.6> Run the program in VTune with input dataset size of 8 KB and 128 KB on an Intel OOO processor. List the number of L1 data cache and L2 cache misses per 1K instructions and the CPI for both configurations. What can you say about the effectiveness of memory latency hiding techniques in high-performance OOO processors? *Hint*: You need to find the L1 data cache miss latency for your processor. For recent Intel i7 processors, it is approximately 11 cycles.

This page intentionally left blank

3.1 Instruction-Level Parallelism: Concepts and Challenges 168
3.2 Basic Compiler Techniques for Exposing ILP 176
3.3 Reducing Branch Costs With Advanced Branch Prediction 182
3.4 Overcoming Data Hazards With Dynamic Scheduling 191
3.5 Dynamic Scheduling: Examples and the Algorithm 201
3.6 Hardware-Based Speculation 208
3.7 Exploiting ILP Using Multiple Issue and Static Scheduling 218
3.8 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation 222
3.9 Advanced Techniques for Instruction Delivery and Speculation 228
3.10 Cross-Cutting Issues 240
3.11 Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput 242
3.12 Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53 247
3.13 Fallacies and Pitfalls 258
3.14 Concluding Remarks: What's Ahead? 264
3.15 Historical Perspective and References 266
Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell 266

3

Instruction-Level Parallelism and Its Exploitation

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."

Dialog between two observers of the sailing race in 1851, later named "The America's Cup," which was the inspiration for John Cocke's naming of an IBM research processor as "America," the first superscalar processor, and a precursor to the PowerPC.

Thus, the IA-64 gambles that, in the future, power will not be the critical limitation, and massive resources…will not penalize clock speed, path length, or CPI factors. My view is clearly skeptical…

Marty Hopkins (2000), IBM Fellow and Early RISC pioneer commenting in 2000 on the new Intel Itanium, a joint development of Intel and HP. The Itanium used a static ILP approach (see Appendix H) and was a massive investment for Intel. It never accounted for more than 0.5% of Intel's microprocessor sales.

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00003-1
© 2019 Elsevier Inc. All rights reserved.

3.1 Instruction-Level Parallelism: Concepts and Challenges

All processors since about 1985 have used pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), because the instructions can be evaluated in parallel. In this chapter and Appendix H, we look at a wide range of techniques for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions.

This chapter is at a considerably more advanced level than the material on basic pipelining in Appendix C. If you are not thoroughly familiar with the ideas in Appendix C, you should review that appendix before venturing into this chapter.

We start this chapter by looking at the limitation imposed by data and control hazards and then turn to the topic of increasing the ability of the compiler and the processor to exploit parallelism. These sections introduce a large number of concepts, which we build on throughout this chapter and the next. While some of the more basic material in this chapter could be understood without all of the ideas in the first two sections, this basic material is important to later sections of this chapter.

There are two largely separable approaches to exploiting ILP: (1) an approach that relies on hardware to help discover and exploit the parallelism dynamically, and (2) an approach that relies on software technology to find parallelism statically at compile time. Processors using the dynamic, hardware-based approach, including all recent Intel and many ARM processors, dominate in the desktop and server markets. In the personal mobile device market, the same approaches are used in processors found in tablets and high-end cell phones. In the IOT space, where power and cost constraints dominate performance goals, designers exploit lower levels of instruction-level parallelism. Aggressive compiler-based approaches have been attempted numerous times beginning in the 1980s and most recently in the Intel Itanium series, introduced in 1999. Despite enormous efforts, such approaches have been successful only in domain-specific environments or in well-structured scientific applications with significant data-level parallelism.

In the past few years, many of the techniques developed for one approach have been exploited within a design relying primarily on the other. This chapter introduces the basic concepts and both approaches. A discussion of the limitations on ILP approaches is included in this chapter, and it was such limitations that directly led to the movement toward multicore. Understanding the limitations remains important in balancing the use of ILP and thread-level parallelism.

In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances.

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

The *ideal pipeline CPI* is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The preceding equation allows us to characterize various techniques by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in Appendix H, as well as the topics covered in the introductory material in Appendix C. In this chapter, we will see that the techniques we introduce to decrease the ideal pipeline CPI can increase the importance of dealing with hazards.

What Is Instruction-Level Parallelism?

All the techniques in this chapter exploit parallelism among instructions. The amount of parallelism available within a *basic block*—a straight-line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. For typical RISC programs, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Because these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level* *parallelism*. Here is a simple example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration, there is little or no opportunity for overlap.

We will examine a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (as in the next section) or dynamically by the hardware (as in Sections 3.5 and 3.6).

An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and graphics processing units (GPUs), both of which are covered in Chapter 4. A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline. For example, the preceding code sequence, which in simple form requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions, might execute in one-quarter as many instructions in some SIMD architecture where four data items are processed per instruction. On some vector processors, this sequence might take only four instructions: two instructions to load the vectors `x` and `y` from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped.

Data Dependences and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism, we must determine which instructions can be executed in parallel. If two instructions are *parallel*, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and thus no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

Data Dependences

There are three different types of dependences: *data dependences* (also called true data dependences), *name dependences*, and *control dependences*. An instruction j is *data-dependent* on instruction i if either of the following holds:

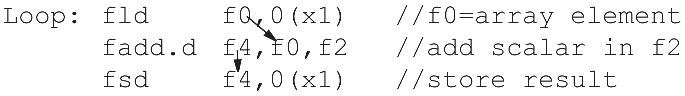
- Instruction *i* produces a result that may be used by instruction *j*.
- Instruction *j* is data-dependent on instruction *k*, and instruction *k* is data-dependent on instruction *i*.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program. Note that a dependence within a single instruction (such as `add x1,x1,x1`) is not considered a dependence.

For example, consider the following RISC-V code sequence that increments a vector of values in memory (starting at `0(x1)` ending with the last element at `0(x2)`) by a scalar in register `f2`.

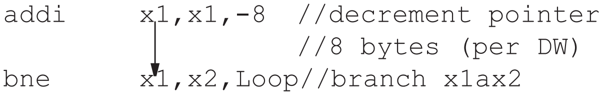
```
Loop:  fld     f0,0(x1)     //f0=array element
       fadd.d  f4,f0,f2     //add scalar in f2
       fsd     f4,0(x1)     //store result
       addi    x1,x1,−8     //decrement pointer 8 bytes
       bne     x1,x2,Loop   //branch x1≠x2
```

The data dependences in this code sequence involve both floating-point data:



```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
```

and integer data:



```
addi     x1,x1,-8  //decrement pointer
                   //8 bytes (per DW)
bne      x1,x2,Loop//branch x1ax2
```

In both of the preceding dependent sequences, as shown by the arrows, each instruction depends on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data-dependent, they must execute in order and cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. (See Appendix C for a brief description of data hazards, which we will define precisely in a few pages.) Executing the instructions simultaneously will cause a processor with pipeline interlocks (and a pipeline depth longer than the distance between the instructions in cycles) to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap because the program will not execute correctly. The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs*. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization*. This difference is critical to understanding how instruction-level parallelism can be exploited.

A data dependence conveys three things: (1) the possibility of a hazard, (2) the order in which results must be calculated, and (3) an upper bound on how much parallelism can possibly be exploited. Such limits are explored in a pitfall on page 262 and in Appendix H in more detail.

Because a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: (1) maintaining the dependence but avoiding a hazard, and (2) eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

A data value may flow between instructions either through registers or through memory locations. When the data flow occurs through a register, detecting the dependence is straightforward because the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns force a compiler or hardware to be conservative.

Dependences that flow through memory locations are more difficult to detect because two addresses may refer to the same location but look different: For example, `100(x4)` and `20(x6)` may be identical memory addresses. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that `20(x4)` and `20(x4)` may be different), further complicating the detection of a dependence.

In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we will see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism.

Name Dependences

The second type of dependence is a *name dependence*. A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction *i* that *precedes* instruction *j* in program order:

1. An *antidependence* between instruction *i* and instruction *j* occurs when instruction *j* writes a register or memory location that instruction *i* reads. The original

ordering must be preserved to ensure that *i* reads the correct value. In the example on page 171, there is an antidependence between `fsd` and `addi` on register `x1`.

2. An *output dependence* occurs when instruction *i* and instruction *j* write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction *j*.

Both antidependences and output dependences are name dependences, as opposed to true data dependences, because there is no value being transmitted between the instructions. Because a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict.

This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Data Hazards

A hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order*—that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program.* Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards, which are informally described in Appendix C, may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions *i* and *j,* with *i* preceding *j* in program order. The possible data hazards are

- RAW *(read after write)*—*j* tries to read a source before *i* writes it, so *j* incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that *j* receives the value from *i*.
- WAW *(write after write)*—*j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

- WAR *(write after read)*—*j* tries to write a destination before it is read by *i,* so *i* incorrectly gets the *new* value. This hazard arises from an antidependence (or name dependence). WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID in the pipeline in Appendix C) and all writes are late (in WB in the pipeline in Appendix C). A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered, as we will see in this chapter.

Note that the RAR *(read after read)* case is not a hazard.

Control Dependences

The last type of dependence is a *control dependence.* A control dependence determines the ordering of an instruction, *i,* with respect to a branch instruction so that instruction *i* is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control-dependent on some set of branches, and in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment

```
if p1 {
      S1;
};
if p2 {
      S2;
}
```

`S1` is control-dependent on `p1`, and `S2` is control-dependent on `p2` but not on `p1`.

In general, two constraints are imposed by control dependences:

1. An instruction that is control-dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control-dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

When processors preserve strict program order, they ensure that control dependences are also preserved. We may be willing to execute instructions that should not have been executed, however, thereby violating the control dependences, *if* we can do so without affecting the correctness of the program. Thus control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness—and normally preserved by maintaining both data and control dependences—are the *exception behavior* and the *data flow*.

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```
        add  x2,x3,x4
        beq  x2,x0,L1
        ld   x1,0(x2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving `x2`, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the `beqz` and the `ld`; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we want to just ignore the exception when the branch is taken. In Section 3.6, we will look at a hardware technique, *speculation*, which allows us to overcome this exception problem. Appendix H looks at software techniques for supporting speculation.

The second property preserved by maintenance of data dependences and control dependences is the data flow. The *data flow* is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic because they allow the source of data for a given instruction to come from many points. Put another way, it is insufficient to just maintain data dependences because an instruction may be data-dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```
        add  x1,x2,x3
        beq  x4,x0,L
        sub  x1,x5,x6
L:      ...
        or   x7,x1,x8
```

In this example, the value of `x1` used by the `or` instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The `or` instruction is data-dependent on both the `add` and `sub` instructions, but preserving that order alone is insufficient for correct execution.

Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken, then the value of `x1` computed by the `sub` should be used by the `or`, and if the branch is taken, the value of `x1` computed by the `add` should be used by the `or`. By preserving the control dependence of the `or` on the branch, we prevent an illegal change to the data flow. For similar reasons, the `sub` instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in Section 3.6.

Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow. Consider the following code sequence:

```
       add  x1,x2,x3
       beq  x12,x0,skip
       sub  x4,x5,x6
       add  x5,x4,x9
skip:  or   x7,x8,x9
```

Suppose we knew that the register destination of the `sub` instruction (`x4`) was unused after the instruction labeled `skip`. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If `x4` were unused, then changing the value of `x4` just before the branch would not affect the data flow because `x4` would be *dead* (rather than live) in the code region after `skip`. Thus, if `x4` were dead and the existing `sub` instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the `sub` instruction before the branch because the data flow could not be affected by this change.

If the branch is taken, the `sub` instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is also a form of speculation, often called software speculation, because the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. More ambitious compiler speculation mechanisms are discussed in Appendix H. Normally, it will be clear when we say speculation or speculative whether the mechanism is a hardware or software mechanism; when it is not clear, it is best to say "hardware speculation" or "software speculation."

Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques, which we examine in Section 3.3.

3.2 Basic Compiler Techniques for Exposing ILP

This section examines the use of simple compiler technology to enhance a processor's ability to exploit ILP. These techniques are crucial for processors that use static issue or static scheduling. Armed with this compiler technology, we will shortly examine the design and performance of processors using static issuing. Appendix H will investigate more sophisticated compiler and associated hardware schemes designed to enable a processor to exploit more instruction-level parallelism.

Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Figure 3.2 shows the FP unit latencies we assume in this chapter, unless different latencies are explicitly stated. We assume the standard five-stage integer pipeline so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth) so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

In this section, we look at how the compiler can increase the amount of available ILP by transforming loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described in Appendix H. We will rely on the following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i−1)
        x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We formalize this notion in Appendix H and describe how we can test whether loop iterations are independent at compile time. First, let's look at the performance of this loop, which shows how we can use the parallelism to improve its performance for a RISC-V pipeline with the preceding latencies.

The first step is to translate the preceding segment to RISC-V assembly language. In the following code segment, `x1` is initially the address of the element in the array with the highest address, and `f2` contains the scalar value *s*. Register `x2` is precomputed so that `Regs[x2]+8` is the address of the last element to operate on.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,−8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for RISC-V with the latencies in Figure 3.2.

Example Show how the loop would look on RISC-V, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                 Clock cycle issued
Loop:  fld     f0,0(x1)                  1
       stall                             2
       fadd.d  f4,f0,f2                  3
       stall                             4
       stall                             5
       fsd     f4,0(x1)                  6
       addi    x1,x1,−8                  7
       bne     x1,x2,Loop                8
```

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

```
Loop:  fld     f0,0(x1)
       addi    x1,x1,−8
       fadd.d  f4,f0,f2
       stall
       stall
       fsd     f4,8(x1)
       bne     x1,x2,Loop
```

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

In the previous example, we complete one loop iteration and store back one array element every seven clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those seven clock cycles.

The remaining four clock cycles consist of loop overhead—the `addi` and `bne`—and two stalls. To eliminate these four clock cycles, we need to get more operations relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus we will want to use different registers for each iteration, increasing the required number of registers.

Example Show our loop unrolled so that there are four copies of the loop body, assuming `x1` − `x2` (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

Answer Here is the result after merging the `addi` instructions and dropping the unnecessary `bne` operations that are duplicated during unrolling. Note that `x2` must now be set so that `Regs[x2]+32` is the starting address of the last four elements.

```
Loop: fld     f0,0(x1)
      fadd.d  f4,f0,f2
      fsd     f4,0(x1)      //drop addi & bne
      fld     f6,−8(x1)
      fadd.d  f8,f6,f2
      fsd     f8,−8(x1)     //drop addi & bne
      fld     f0,−16(x1)
      fadd.d  f12,f0,f2
      fsd     f12,−16(x1)   //drop addi & bne
      fld     f14,−24(x1)
      fadd.d  f16,f14,f2
      fsd     f16,−24(x1)
      addi    x1,x1,−32
      bne     x1,x2,Loop
```

We have eliminated three branches and three decrements of `x1`. The addresses on the loads and stores have been compensated to allow the `addi` instructions on `x1` to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. Symbolic substitution and simplification will rearrange expressions so as to allow constants to be collapsed, allowing an expression such as $((i+1)+1)$ to be rewritten as $(i+(1+1))$ and then simplified to $(i+2)$.

We will see more general forms of these optimizations that eliminate dependent computations in Appendix H.

Without scheduling, every FP load or operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This unrolled loop will run in 26 clock cycles—each `fld` has 1 stall, each `fadd.d` has 2, plus 14 instruction issue cycles—or 6.5 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly. Loop unrolling is normally done early in the compilation process so that redundant computations can be exposed and eliminated by the optimizer.

In real programs, we do not usually know the upper bound on the loop. Suppose it is *n,* and we want to unroll the loop to make *k* copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (*n* mod *k*) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times. (As we will see in Chapter 4, this technique is similar to a technique called *strip mining*, used in compilers for vector processors.) For large values of *n*, most of the execution time will be spent in the unrolled loop body.

In the previous example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

Example Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies in Figure 3.2.

Answer

```
Loop:  fld     f0,0(x1)
       fld     f6,−8(x1)
       fld     f0,−16(x1)
       fld     f14,−24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,−8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,−32
       bne     x1,x2,Loop
```

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 8 cycles per element before any unrolling or scheduling and 6.5 cycles when unrolled but not scheduled.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the preceding code has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of the Loop Unrolling and Scheduling

Throughout this chapter and Appendix H, we will look at a variety of hardware and software techniques that allow us to take advantage of instruction-level parallelism to fully utilize the potential of the functional units in a processor. The key to most of these techniques is to know when and how the ordering among instructions may be changed. In our example, we made many such changes, which to us, as human beings, were obviously allowable. In practice, this process must be performed in a methodical fashion either by a compiler or by hardware. To obtain the final unrolled code, we had to make the following decisions and transformations:

- Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
- Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations (e.g., name dependences).
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how one instruction depends on another and how the instructions can be changed or reordered given the dependences.

Three different effects limit the gains from loop unrolling: (1) a decrease in the amount of overhead amortized with each unroll, (2) code size limitations, and (3) compiler limitations. Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. In fact, in 14 clock cycles, only 2 cycles were loop overhead: the `addi`, which maintains the index value, and the `bne`, which terminates the loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per element to 1/4.

A second limit to unrolling is the resulting growth in code size. For larger loops, the code size growth may be a concern, particularly if it causes an increase in the instruction cache miss rate.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect that results from instruction scheduling in large code segments is called *register pressure*. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage because it leads to a shortage of registers. Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem. The combination of unrolling and aggressive scheduling can, however, cause this problem. The problem becomes especially challenging in multiple-issue processors that require the exposure of more independent instruction sequences whose execution can be overlapped. In general, the use of sophisticated high-level transformations, whose potential improvements are difficult to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Loop unrolling is a simple but useful method for increasing the size of straight-line code fragments that can be scheduled effectively. This transformation is useful in a variety of processors, from simple pipelines like those we have examined so far to the multiple-issue superscalars and VLIWs explored later in this chapter.

3.3 Reducing Branch Costs With Advanced Branch Prediction

Because of the need to enforce control dependences through branch hazards and stalls, branches will hurt pipeline performance. Loop unrolling is one way to reduce the number of branch hazards; we can also reduce the performance losses of branches by predicting how they will behave. In Appendix C, we examine simple branch predictors that rely either on compile-time information or on the observed dynamic behavior of a single branch in isolation. As the number of instructions in flight has increased with deeper pipelines and more issues per clock, the importance of more accurate branch prediction has grown. In this section, we examine techniques for improving dynamic prediction accuracy. This section makes extensive use of the simple 2-bit predictor covered in Section C.2, and it is critical that the reader understand the operation of that predictor before proceeding.

Correlating Branch Predictors

The 2-bit predictor schemes in Appendix C use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of *other* branches rather than just the branch we are trying to predict. Consider a small code fragment from the eqntott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

```
if (aa==2)
      aa=0;
if (bb==2)
      bb=0;
if (aa!=bb) {
```

Here is the RISC-V code that we would typically generate for this code fragment assuming that `aa` and `bb` are assigned to registers `x1` and `x2`:

```
    addi  x3,x1,-2
    bnez  x3,L1        //branch b1  (aa!=2)
    add   x1,x0,x0     //aa=0
L1: addi  x3,x2,-2
    bnez  x3,L2        //branch b2  (bb!=2)
    add   x2,x0,x0     //bb=0
L2: sub   x3,x1,x2     //x3=aa-bb
    beqz  x3,L3        //branch b3  (aa==bb)
```

Let's label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if neither branches b1 nor b2 are taken (i.e., if the conditions both evaluate to true and `aa` and `bb` are both assigned 0), then b3 will be taken, because `aa` and `bb` are clearly equal. A predictor that uses the behavior of only a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. In the general case, an (*m,n*) predictor uses the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the 2-bit scheme and requires only a trivial amount of additional hardware.

The simplicity of the hardware comes from a simple observation: the global history of the most recent *m* branches can be recorded in an *m*-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the *m*-bit global history. For example, in a (2,2) buffer with 64 total entries, the 4 low-order address bits of the branch (word address) and the 2 global bits representing the behavior of the two most recently executed branches form a 6-bit index that can be used to index the 64 counters. By combining the local and global information by concatenation (or a simple hash function), we can index the predictor table with the result and get a prediction as fast as we could for the standard 2-bit predictor, as we will do very shortly.

How much better do the correlating branch predictors work when compared with the standard 2-bit scheme? To compare them fairly, we must compare predictors that use the same number of state bits. The number of bits in an (m,n) predictor is

$$2^m \times n \times \text{Number of prediction entries selected by the branch address}$$

A 2-bit predictor with no global history is simply a (0,2) predictor.

Example How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer The predictor with 4K entries has

$$2^0 \times 2 \times 4\text{K} = 8\text{K bits}$$

How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer? We know that

$$2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8\text{K}$$

Therefore the number of prediction entries selected by the branch = 1K.

Figure 3.3 compares the misprediction rates of the earlier (0,2) predictor with 4K entries and a (2,2) predictor with 1K entries. As you can see, this correlating predictor not only outperforms a simple 2-bit predictor with the same total number of state bits, but it also often outperforms a 2-bit predictor with an unlimited number of entries.

Perhaps the best-known example of a correlating predictor is McFarling's gshare predictor. In gshare the index is formed by combining the address of the branch and the most recent conditional branch outcomes using an exclusive-OR, which essentially acts as a hash of the branch address and the branch history. The hashed result is used to index a prediction array of 2-bit counters, as shown in Figure 3.4. The gshare predictor works remarkably well for a simple predictor, and is often used as the baseline for comparison with more sophisticated predictors. Predictors that combine local branch information and global branch history are also called *alloyed predictors or hybrid predictors*.

Tournament Predictors: Adaptively Combining Local and Global Predictors

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor, using only local information, failed on some important branches. Adding global history could help remedy this situation. *Tournament predictors* take this insight to the next level, by using multiple predictors, usually a global predictor and a local predictor, and choosing between them

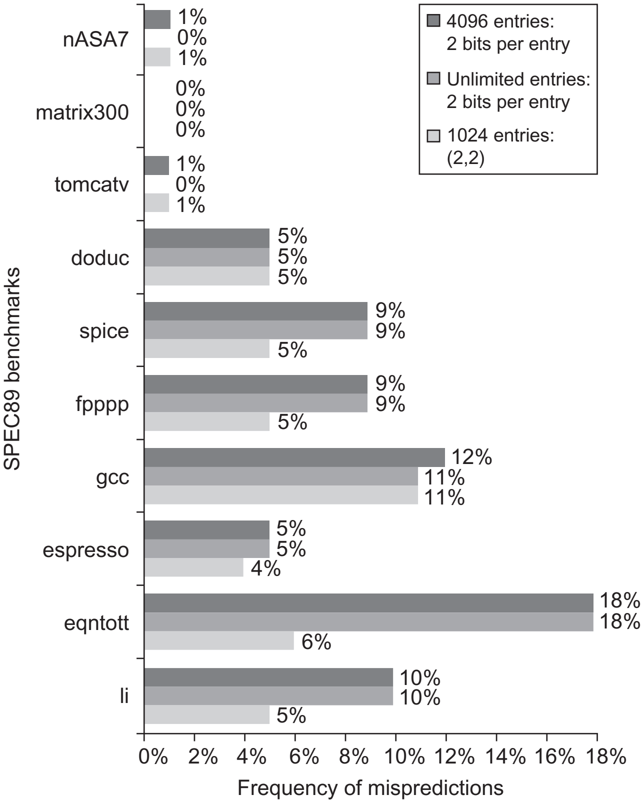


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

with a selector, as shown in Figure 3.5. A *global predictor* uses the most recent branch history to index the predictor, while a *local predictor* uses the address of the branch as the index. Tournament predictors are another form of hybrid or alloyed predictors.

Tournament predictors can achieve better accuracy at medium sizes (8K–32K bits) and also effectively use very large numbers of prediction bits. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some time-varying mix) was most effective in recent predictions. As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred predictor.

The advantage of a tournament predictor is its ability to select the right predictor for a particular branch, which is particularly crucial for the integer benchmarks.

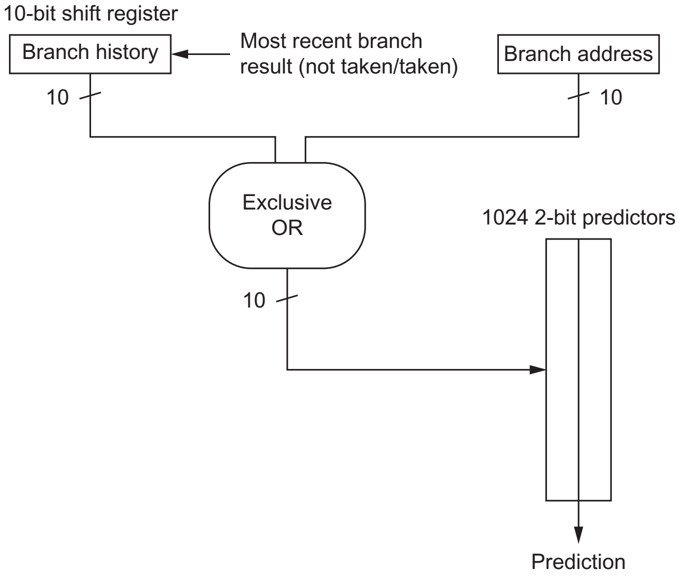

10-bit shift register
Branch history
Most recent branch result (not taken/taken)
Branch address
10
10
Exclusive OR
1024 2-bit predictors
10
Prediction

Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

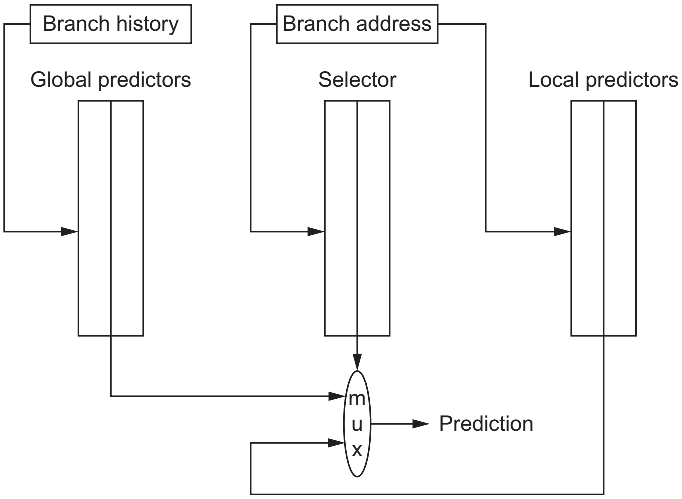

Branch history
Branch address
Global predictors
Selector
Local predictors
m
u
x
Prediction

Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks. In addition to the Alpha processors that pioneered tournament predictors, several AMD processors have used tournament-style predictors.

Figure 3.6 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. The local predictor reaches its limit first. The correlating predictor shows a significant improvement, and the tournament predictor generates a slightly better performance. For more recent versions of the SPEC, the results would be similar, but the asymptotic behavior would not be reached until slightly larger predictor sizes.

The local predictor consists of a two-level predictor. The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. That is, if the branch is taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken, the history entry consists of alternating 0s and 1s. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29K bits, leads to high accuracy in

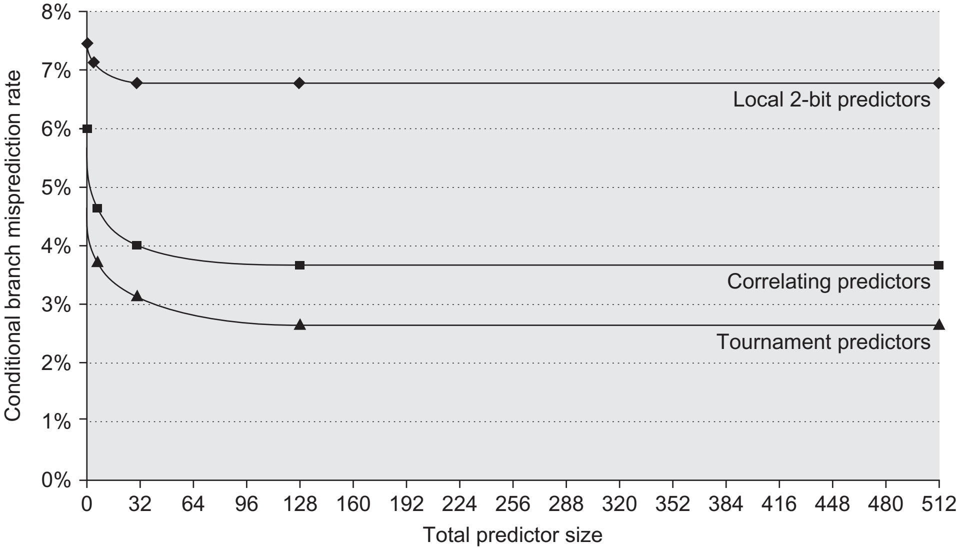


Figure 3.6 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

branch prediction while requiring fewer bits than a single level table with the same prediction accuracy.

Tagged Hybrid Predictors

The best performing branch prediction schemes as of 2017 involve combining multiple predictors that track whether a prediction is likely to be associated with the current branch. One important class of predictors is loosely based on an algorithm for statistical compression called PPM (Prediction by Partial Matching). PPM (see Jiménez and Lin, 2001), like a branch prediction algorithm, attempts to predict future behavior based on history. This class of branch predictors, which we call *tagged hybrid predictors* (see Seznec and Michaud, 2006), employs a series of global predictors indexed with different length histories.

For example, as shown in Figure 3.7, a five-component tagged hybrid predictor has five prediction tables: P(0), P(1), . . . P(4), where P(i) is accessed using a hash of

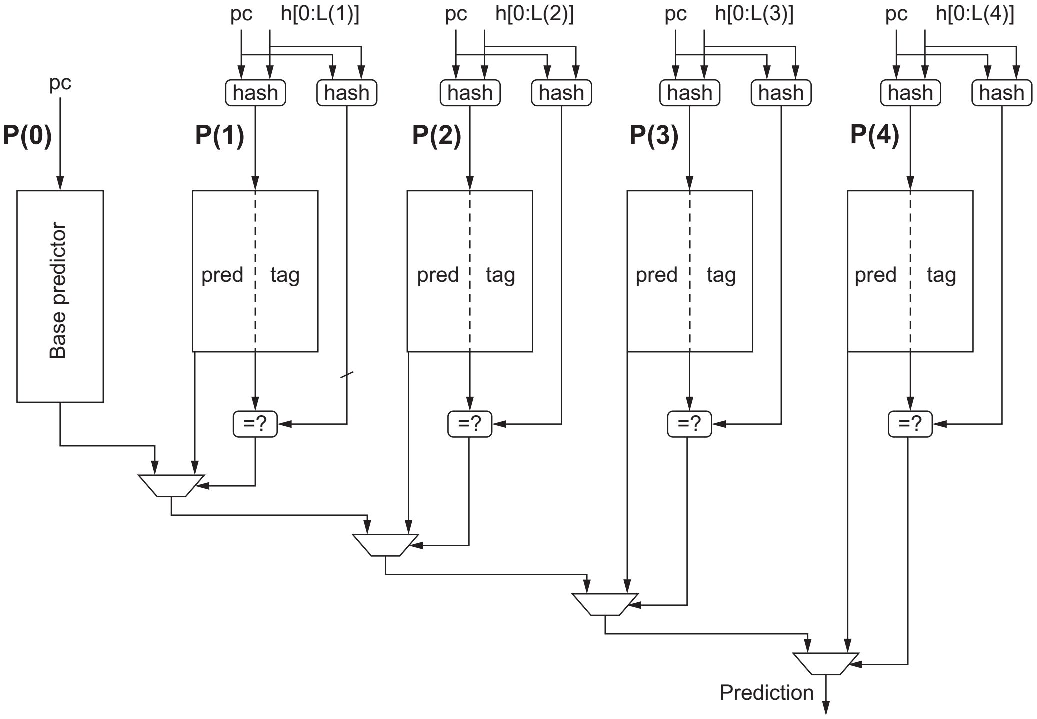


Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

the PC and the history of the most recent *i* branches (kept in a shift register, h, just as in gshare). The use of multiple history lengths to index separate predictors is the first critical difference. The second critical difference is the use of tags in tables P(1) through P(4). The tags can be short because 100% matches are not required: a small tag of 4–8 bits appears to gain most of the advantage. A prediction from P(1), . . . P(4) is used only if the tags match the hash of the branch address and global branch history. Each of the predictors in P(0…*n*) can be a standard 2-bit predictor. In practice a 3-bit counter, which requires three mispredictions to change a prediction, gives slightly better results than a 2-bit counter.

The prediction for a given branch is the predictor with the longest branch history that also has matching tags. P(0) always matches because it uses no tags and becomes the default prediction if none of P(1) through P(*n*) match. The tagged hybrid version of this predictor also includes a 2-bit use field in each of the history-indexed predictors. The use field indicates whether a prediction was recently used and therefore likely to be more accurate; the use field can be periodically reset in all entries so that old predictions are cleared. Many more details are involved in implementing this style of predictor, especially how to handle mispredictions. The search space for the optimal predictor is also very large because the number of predictors, the exact history used for indexing, and the size of each predictor are all variable.

Tagged hybrid predictors (sometimes called TAGE—TAgged GEometic—predictors) and the earlier PPM-based predictors have been the winners in recent annual international branch-prediction competitions. Such predictors outperform gshare and the tournament predictors with modest amounts of memory (32–64 KiB), and in addition, this class of predictors seems able to effectively use larger prediction caches to deliver improved prediction accuracy.

Another issue for larger predictors is how to initialize the predictor. It could be initialized randomly, in which case, it will take a fair amount of execution time to fill the predictor with useful predictions. Some predictors (including many recent predictors) include a valid bit, indicating whether an entry in the predictor has been set or is in the "unused state." In the latter case, rather than use a random prediction, we could use some method to initialize that prediction entry. For example, some instruction sets contain a bit that indicates whether an associated branch is expected to be taken or not. In the days before dynamic branch prediction, such hint bits *were* the prediction; in recent processors, that hint bit can be used to set the initial prediction. We could also set the initial prediction on the basis of the branch direction: forward going branches are initialized as not taken, while backward going branches, which are likely to be loop branches, are initialized as taken. For programs with shorter running times and processors with larger predictors, this initial setting can have a measurable impact on prediction performance.

Figure 3.8 shows that a hybrid tagged predictor significantly outperforms gshare, especially for the less predictable programs like SPECint and server applications. In this figure, performance is measured as mispredicts per thousand instructions; assuming a branch frequency of 20%–25%, gshare has a mispredict rate (per branch) of 2.7%–3.4% for the multimedia benchmarks, while the tagged

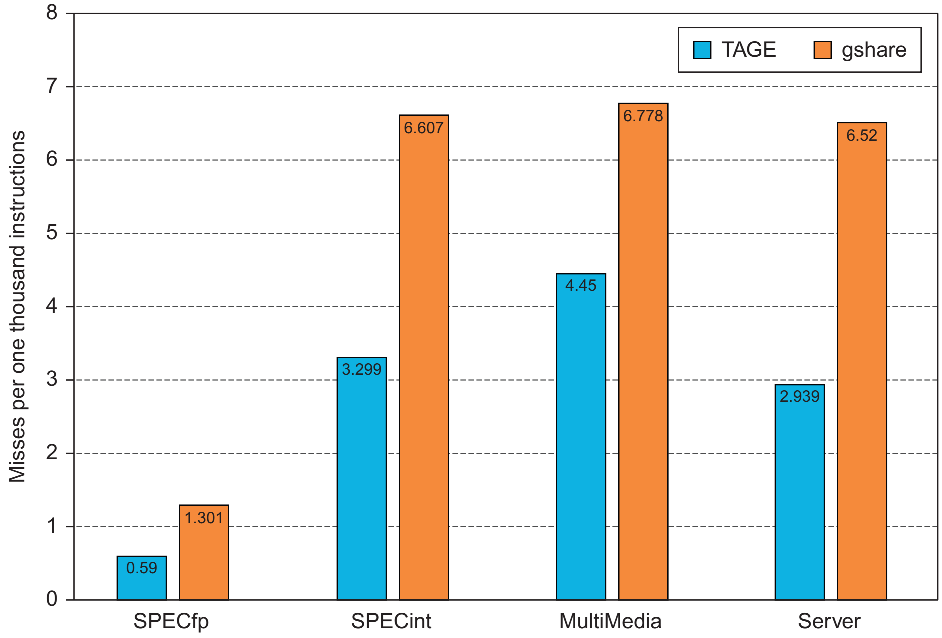


Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

hybrid predictor has a misprediction rate of 1.8%–2.2%, or roughly one-third fewer mispredicts. Compared to gshare, tagged hybrid predictors are more complex to implement and are probably slightly slower because of the need to check multiple tags and choose a prediction result. Nonetheless, for deeply pipelined processors with large penalties for branch misprediction, the increased accuracy outweighs those disadvantages. Thus many designers of higher-end processors have opted to include tagged hybrid predictors in their newest implementations.

The Evolution of the Intel Core i7 Branch Predictor

As mentioned in the previous chapter, there were six generations of Intel Core i7 processors between 2008 (Core i7 920 using the Nehalem microarchitecture) and 2016 (Core i7 6700 using the Skylake microarchitecture). Because of the combination of deep pipelining and multiple issues per clock, the i7 has many instructions in-flight at once (up to 256, and typically at least 30). This makes branch prediction critical, and it has been an area where Intel has been making constant improvements. Perhaps because of the performance-critical nature of the branch predictor, Intel has tended to keep the details of its branch predictors highly secret.

Even for older processors such as the Core i7 920 introduced in 2008, they have released only limited amounts of information. In this section, we briefly describe what is known and compare the performance of predictors of the Core i7 920 with those in the latest Core i7 6700.

The Core i7 920 used a two-level predictor that has a smaller first-level predictor, designed to meet the cycle constraints of predicting a branch every clock cycle, and a larger second-level predictor as a backup. Each predictor combines three different predictors: (1) the simple 2-bit predictor, which is introduced in Appendix C (and used in the preceding tournament predictor); (2) a global history predictor, like those we just saw; and (3) a loop exit predictor. The loop exit predictor uses a counter to predict the exact number of taken branches (which is the number of loop iterations) for a branch that is detected as a loop branch. For each branch, the best prediction is chosen from among the three predictors by tracking the accuracy of each prediction, like a tournament predictor. In addition to this multilevel main predictor, a separate unit predicts target addresses for indirect branches, and a stack to predict return addresses is also used.

Although even less is known about the predictors in the newest i7 processors, there is good reason to believe that Intel is employing a tagged hybrid predictor. One advantage of such a predictor is that it combines the functions of all three second-level predictors in the earlier i7. The tagged hybrid predictor with different history lengths subsumes the loop exit predictor as well as the local and global history predictor. A separate return address predictor is still employed.

As in other cases, speculation causes some challenges in evaluating the predictor because a mispredicted branch can easily lead to another branch being fetched and mispredicted. To keep things simple, we look at the number of mispredictions as a percentage of the number of successfully completed branches (those that were not the result of misspeculation). Figure 3.9 shows these data for SPECPUint2006 benchmarks. These benchmarks are considerably larger than SPEC89 or SPEC2000, with the result being that the misprediction rates are higher than those in Figure 3.6 even with a more powerful combination of predictors. Because branch misprediction leads to ineffective speculation, it contributes to the wasted work, as we will see later in this chapter.

3.4 Overcoming Data Hazards With Dynamic Scheduling

A simple statically scheduled pipeline fetches an instruction and issues it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. (Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards.) If there is a data dependence that cannot be hidden, then the hazard detection hardware stalls the pipeline starting with the instruction that uses the result. No new instructions are fetched or issued until the dependence is cleared.

In this section, we explore *dynamic scheduling*, a technique by which the hardware reorders the instruction execution to reduce the stalls while maintaining data

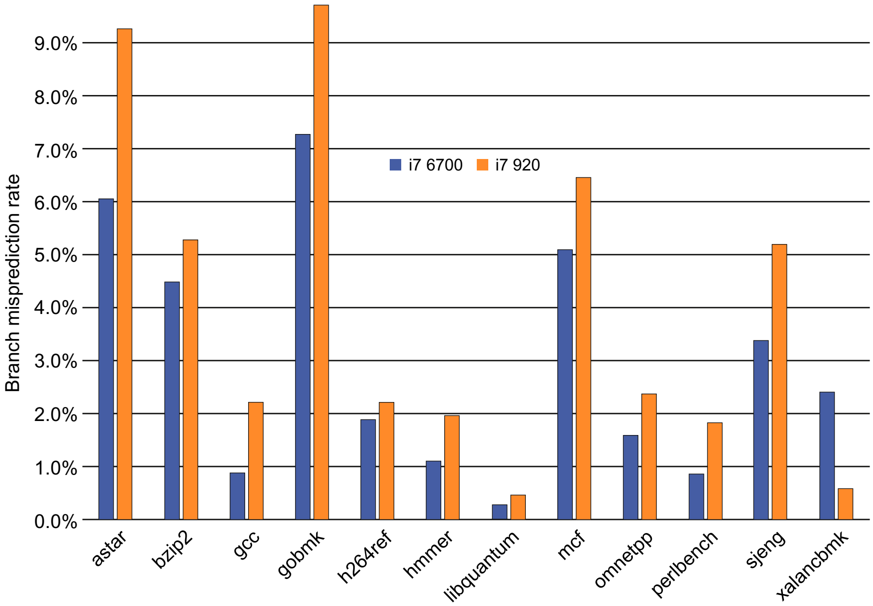


Figure 3.9 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 920 and 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. This could understate the misprediction rate somewhat because if a branch is mispredicted and led to another mispredicted branch (which should not have been executed), it will be counted as only one misprediction. On average, the i7 920 mispredicts branches 1.3 times as often as the i7 6700.

flow and exception behavior. Dynamic scheduling offers several advantages. First, it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, eliminating the need to have multiple binaries and recompile for a different microarchitecture. In today's computing environment, where much of the software is from third parties and distributed in binary form, this advantage is significant. Second, it enables handling some cases when dependences are unknown at compile time; for example, they may involve a memory reference or a data-dependent branch, or they may result from a modern programming environment that uses dynamic linking or dispatching. Third, and perhaps most importantly, it allows the processor to tolerate unpredictable delays, such as cache misses, by executing other code while waiting for the miss to resolve. In Section 3.6, we explore hardware speculation, a technique with additional performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at the cost of a significant increase in hardware complexity.

Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences are present. In contrast, static pipeline scheduling by the compiler (covered in Section 3.2) tries to minimize stalls by separating dependent instructions so that they will not lead to hazards. Of course, compiler pipeline scheduling can also be used in code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

A major limitation of simple pipelining techniques is that they use in-order instruction issue and execution: instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, it will lead to a hazard, and a stall will result. If there are multiple functional units, these units could lie idle. If instruction *j* depends on a long-running instruction *i,* currently in execution in the pipeline, then all instructions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

```
fdiv.d  f0,f2,f4
fadd.d  f10,f0,f8
fsub.d  f12,f8,f14
```

The `fsub.d` instruction cannot execute because the dependence of `fadd.d` on `fdiv.d` causes the pipeline to stall; yet, `fsub.d` is not data-dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline, both structural and data hazards could be checked during instruction decode (ID): when an instruction could execute without hazards, it was issued from ID, with the recognition that all data hazards had been resolved.

To allow us to begin executing the `fsub.d` in the preceding example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. Thus we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does *out-of-order execution*, which implies *out-of-order completion*.

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following RISC-V floating-point code sequence:

```
fdiv.d  f0,f2,f4
fmul.d  f6,f0,f8
fadd.d  f0,f10,f14
```

There is an antidependence between the `fmul.d` and the `fadd.d` (for the register `f0`), and if the pipeline executes the `fadd.d` before the `fmul.d` (which is waiting for the `fdiv.d`), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of `f0` by `fadd.d` before `fdiv.d` completes, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise. Dynamically scheduled processors preserve exception behavior by delaying the notification of an associated exception until the processor knows that the instruction should be the next one completed.

Although exception behavior must be preserved, dynamically scheduled processors could generate *imprecise* exceptions. An exception is *imprecise* if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have *already completed* instructions that are *later* in program order than the instruction causing the exception.
2. The pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation in Section 3.6. For floating-point exceptions, other solutions have been used, as discussed in Appendix J.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

1. *Issue*—Decode instructions, check for structural hazards.
2. *Read operands*—Wait until no data hazards, then read operands.

An instruction fetch stage precedes the issue stage and may fetch either to an instruction register or into a queue of pending instructions; instructions are then issued from the register or queue. The execution stage follows the read operands stage, just as in the five-stage pipeline. Execution may take multiple cycles, depending on the operation.

We distinguish when an instruction *begins execution* and when it *completes execution*; between the two times, the instruction is *in execution.* Our pipeline allows multiple instructions to be in execution at the same time; without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or both. Because these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or can bypass each other in the second stage (read operands) and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability. Here we focus on a more sophisticated technique, called *Tomasulo's algorithm.* The primary difference is that Tomasulo's algorithm handles antidependences and output dependences by effectively renaming the registers dynamically. Additionally, Tomasulo's algorithm can be extended to handle *speculation*, a technique to reduce the effect of control dependences by predicting the outcome of a branch, executing instructions at the predicted destination address, and taking corrective actions when the prediction was wrong. While the use of scoreboarding is probably sufficient to support simpler processors, more sophisticated, higher performance processors make use of speculation.

Dynamic Scheduling Using Tomasulo's Approach

The IBM 360/91 floating-point unit used a sophisticated scheme to allow out-of-order execution. This scheme, invented by Robert Tomasulo, tracks when operands for instructions are available to minimize RAW hazards and introduces register renaming in hardware to minimize WAW and WAR hazards. Although there are many variations of this scheme in recent processors, they all rely on two key principles: dynamically determining when an instruction is ready to execute and renaming registers to avoid unnecessary hazards.

IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360 computer family, rather than from specialized compilers for the high-end processors. The 360 architecture had only four double-precision floating-point registers, which limited the effectiveness of compiler scheduling; this fact was another motivation for the Tomasulo approach. In addition, the IBM 360/91 had long memory accesses and long floating-point delays, which Tomasulo's algorithm was designed to overcome. At the end of the section, we will see that Tomasulo's algorithm can also support the overlapped execution of multiple iterations of a loop.

We explain the algorithm, which focuses on the floating-point unit and load-store unit, in the context of the RISC-V instruction set. The primary difference between RISC-V and the 360 is the presence of register-memory instructions in the latter architecture. Because Tomasulo's algorithm uses a load functional unit, no significant changes are needed to add register-memory addressing modes. The IBM 360/91 also had pipelined functional units, rather than multiple functional units, but we describe the algorithm as if there were multiple functional units. It is a simple conceptual extension to also pipeline those functional units.

RAW hazards are avoided by executing an instruction only when its operands are available, which is exactly what the simpler scoreboarding approach provides. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. *Register renaming* eliminates these hazards by renaming all destination registers, including those with a pending read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand. The compiler could typically implement such renaming, if there were enough registers available in the ISA. The original 360/91 had only four floating-point registers, and Tomasulo's algorithm was created to overcome this shortage. Whereas modern processors have 32–64 floating-point and integer registers, the number of renaming registers available in recent implementations is in the hundreds.

To better understand how register renaming eliminates WAR and WAW hazards, consider the following example code sequence that includes potential WAR and WAW hazards:

```
fdiv.d  f0,f2,f4
fadd.d  f6,f0,f8
fsd     f6,0(x1)
fsub.d  f8,f10,f14
fmul.d  f6,f10,f8
```

There are two antidependences: between the `fadd.d` and the `fsub.d` and between the `fsd` and the `fmul.d`. There is also an output dependence between the `fadd.d` and the `fmul.d`, leading to three possible hazards: WAR hazards on the use of `f8` by `fadd.d` and its use by the `fsub.d`, as well as a WAW hazard because the `fadd.d` may finish later than the `fmul.d`. There are also three true data dependences: between the `fdiv.d` and the `fadd.d`, between the `fsub.d` and the `fmul.d`, and between the `fadd.d` and the `fsd`.

These three name dependences can all be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, `S` and `T`. Using `S` and `T`, the sequence can be rewritten without any dependences as

```
fdiv.d  f0,f2,f4
fadd.d  S,f0,f8
fsd     S,0(x1)
fsub.d  T,f10,f14
fmul.d  f6,f10,T
```

In addition, any subsequent uses of `f8` must be replaced by the register `T`. In this example, the renaming process can be done statically by the compiler. Finding any uses of `f8` that are later in the code requires either sophisticated compiler analysis or hardware support because there may be intervening branches between the preceding code segment and a later use of `f8`. As we will see, Tomasulo's algorithm can handle renaming across branches.

In Tomasulo's scheme, register renaming is provided by *reservation stations*, which buffer the operands of instructions waiting to issue and are associated with the functional units. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming.

Because there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards.

The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: the information held in the reservation stations at each functional unit determines when an instruction can begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91, this is called the *common data bus*, or CDB). In pipelines that issue multiple instructions per clock and also have multiple execution units, more than one result bus will be needed.

Figure 3.10 shows the basic structure of a Tomasulo-based processor, including both the floating-point unit and the load/store unit; none of the execution control tables is shown. Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit. If the operand values for that instruction have been computed, they are also stored in that entry; otherwise, the reservation station entry keeps the names of the reservation stations that will provide the operand values.

The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations, so we distinguish them only when necessary. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All reservation stations have tag fields, employed by the pipeline control.

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through. There are only three steps, although each one can now take an arbitrary number of clock cycles:

1. *Issue*—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard, and the instruction issue stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards. (This stage is sometimes called *dispatch* in a dynamically scheduled processor.)

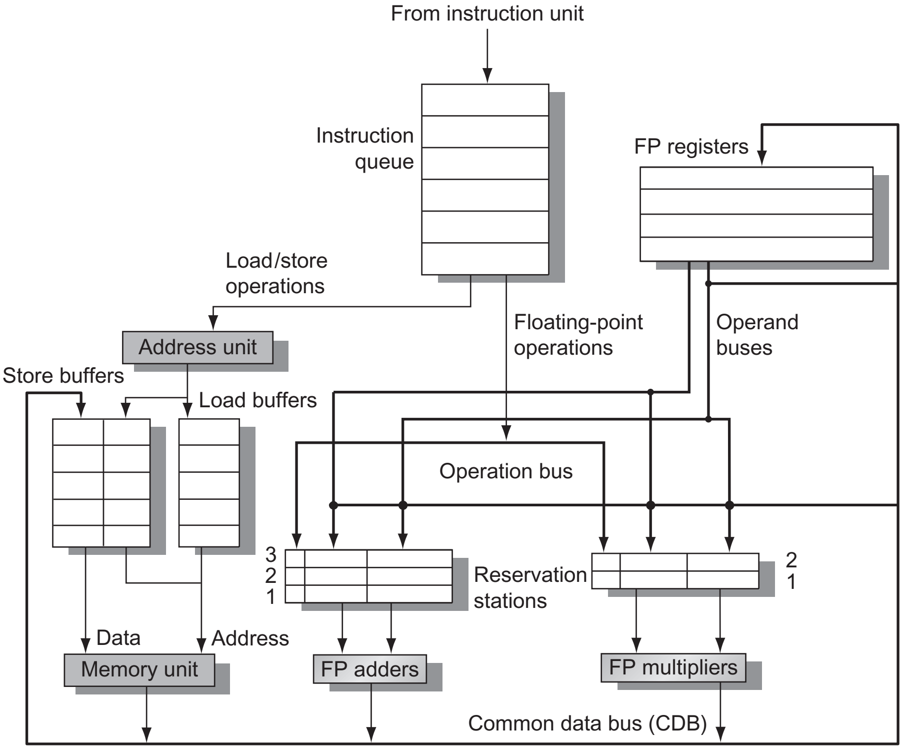

From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: (1) hold the components of the effective address until it is computed, (2) track outstanding loads that are waiting on the memory, and (3) hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: (1) hold the components of the effective address until it is computed, (2) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (3) hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

2. *Execute*—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided. (Some dynamically scheduled processors call this step "issue," but we use the name "execute," which was used in the first dynamically scheduled processor, the CDC 6600.)

Notice that several instructions could become ready in the same clock cycle for the same functional unit. Although independent functional units could begin execution in the same clock cycle for different instructions, if more than one instruction is ready for a single functional unit, the unit will have to choose among them. For the floating-point reservation stations, this choice may be made arbitrarily; loads and stores, however, present an additional complication.

Loads and stores require a two-step execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit. Loads and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory.

To preserve exception behavior, no instruction is allowed to initiate execution until a branch that precedes the instruction in program order has completed. This restriction guarantees that an instruction that causes an exception during execution really would have been executed. In a processor using branch prediction (as all dynamically scheduled processors do), this means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution. If the processor records the occurrence of the exception, but does not actually raise it, an instruction can start execution but not stall until it enters Write Result.

Speculation provides a more flexible and more complete method to handle exceptions, so we will delay making this enhancement and show how speculation handles this problem later.

3. *Write result*—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores are buffered in the store buffer until both the value to be stored and the store address are available; then the result is written as soon as the memory unit is free.

The data structures that detect and eliminate hazards are attached to the reservation stations, to the register file, and to the load and store buffers with slightly different information attached to different objects. These tags are essentially names for an extended set of virtual registers used for renaming. In our example, the tag field is a 4-bit quantity that denotes one of the five reservation stations or one of the five load buffers. This combination produces the equivalent of 10 registers (5 reservation stations + 5 load buffers) that can be designated as result registers (as opposed to the four double-precision registers that the 360 architecture contains). In a processor with more real registers, we want renaming to provide an even larger set of virtual registers, often numbering in the hundreds. The tag field describes which reservation station contains the instruction that will produce a result needed as a source operand.

Once an instruction has issued and is waiting for a source operand, it refers to the operand by the reservation station number where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in the registers. Because there are more reservation stations than actual register numbers, WAW and WAR hazards are eliminated by renaming results using reservation station numbers. Although in Tomasulo's scheme the reservation stations are used as the extended virtual registers, other approaches could use a register set with additional registers or a structure like the reorder buffer, which we will see in Section 3.6.

In Tomasulo's scheme, as well as the subsequent methods we look at for supporting speculation, results are broadcast on a bus (the CDB), which is monitored by the reservation stations. The combination of the common result bus and the retrieval of results from the bus by the reservation stations implements the forwarding and bypassing mechanisms used in a statically scheduled pipeline. In doing so, however, a dynamically scheduled scheme, such as Tomasulo's algorithm, introduces one cycle of latency between source and result because the matching of a result and its use cannot be done until the end of the Write Result stage, as opposed to the end of the Execute stage for a simpler pipeline. Thus, in a dynamically scheduled pipeline, the effective latency between a producing instruction and a consuming instruction is at least one cycle longer than the latency of the functional unit producing the result.

It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register names are discarded when an instruction issues to a reservation station. (This is a key difference between Tomasulo's scheme and scoreboarding: in scoreboarding, operands stay in the registers and are read only after the producing instruction completes and the consuming instruction is ready to execute.)

Each reservation station has seven fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk—The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
- A—Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

The register file has a field, Qi:

- Qi—The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

The load and store buffers each have a field, A, which holds the result of the effective address once the first step of execution has been completed.

In the next section, we will first consider some examples that show how these mechanisms work and then examine the detailed algorithm.

3.5 Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider a few examples that will help illustrate how the algorithm works.

Example Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

```
1.  fld     f6,32(x2)
2.  fld     f2,44(x3)
3.  fmul.d  f0,f2,f4
4.  fsub.d  f8,f2,f6
5.  fdiv.d  f0,f0,f6
6.  fadd.d  f6,f8,f2
```

Answer Figure 3.11 shows the result in three tables. The numbers appended to the names Add, Mult, and Load stand for the tag for that reservation station—Add1 is the tag for the result from the first add unit. In addition, we have included an instruction status table. This table is included only to help you understand the algorithm; it is *not* actually a part of the hardware. Instead, the reservation station keeps the state of each operation that has issued.

Tomasulo's scheme offers two major advantages over earlier and simpler schemes: (1) the distribution of the hazard detection logic, and (2) the elimination of stalls for WAW and WAR hazards.

The first advantage arises from the distributed reservation stations and the use of the CDB. If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast of the result on the CDB. If a centralized register file were used, the units would have to read their results from the registers when register buses were available.

The second advantage, the elimination of WAW and WAR hazards, is accomplished by renaming registers using the reservation stations and by the process of storing operands into the reservation station as soon as they are available.

For example, the code sequence in Figure 3.11 issues both the `fdiv.d` and the `fadd.d`, even though there is a WAR hazard involving `f6`. The hazard is eliminated in one of two ways. First, if the instruction providing the value for the `fdiv.d` has completed, then Vk will store the result, allowing `fdiv.d` to execute independent of the `fadd.d` (this is the case shown). On the other hand, if the `fld` hasn't completed, then Qk will point to the Load1 reservation station, and the `fdiv.d` instruction will be independent of the `fadd.d`. Thus, in either case, the `fadd.d` can issue and begin executing. Any uses of the result of the `fdiv.d` will point to the reservation station, allowing the `fadd.d` to complete and store its value into the registers without affecting the `fdiv.d`.

	Instruction status		
Instruction	**Issue**	**Execute**	**Write result**
`fld f6,32(x2)`	√	√	√
`fld f2,44(x3)`	√	√	
`fmul.d f0,f2,f4`	√		
`fsub.d f8,f2,f6`	√		
`fdiv.d f0,f0,f6`	√		
`fadd.d f6,f8,f2`	√		

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	No						
Load2	Yes	Load					`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		`Regs[f4]`	Load2		
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status							
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	**…**	**f30**
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.11 Reservation stations and register tags shown when all of the instructions have issued but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the `fadd.d` instruction, which has a WAR hazard at the WB stage, has issued and could complete before the `fdiv.d` initiates.

We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution. In this example, and the ones that follow in this chapter, assume the following latencies: load is 1 clock cycle, add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.

Example Using the same code segment as in the previous example (page 201), show what the status tables look like when the `fmul.d` is ready to write its result.

Answer The result is shown in the three tables in Figure 3.12. Notice that `fadd.d` has completed because the operands of `fdiv.d` were copied, thereby overcoming the WAR hazard. Notice that even if the load of `f6` was `fdiv.d`, the add into `f6` could be executed without triggering a WAW hazard.

	Instruction status		
Instruction	**Issue**	**Execute**	**Write result**
`fld f6,32(x2)`	√	√	√
`fld f2,44(x3)`	√	√	√
`fmul.d f0,f2,f4`	√	√	
`fsub.d f8,f2,f6`	√	√	√
`fdiv.d f0,f0,f6`	√		
`fadd.d f6,f8,f2`	√	√	√

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	`Mem[44 + Regs[x3]]`	`Regs[f4]`			
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status							
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	**…**	**f30**
Qi	Mult1					Mult2			

Figure 3.12 Multiply and divide are the only instructions not finished.

Tomasulo's Algorithm: The Details

Figure 3.13 specifies the checks and steps that each instruction must go through. As mentioned earlier, loads and stores go through a functional unit for effective address computation before proceeding to independent load or store buffers. Loads take a second execution step to access memory and then go to Write Result to send the value from memory to the register file and/or any waiting reservation stations. Stores complete their execution in the Write Result stage, which writes the result to memory. Notice that all writes occur in Write Result, whether the destination is a register or memory. This restriction simplifies Tomasulo's algorithm and is critical to its extension with speculation in Section 3.6.

Tomasulo's Algorithm: A Loop-Based Example

To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the following simple sequence for multiplying the elements of an array by a scalar in `f2`:

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,−8
       bne     x1,x2,Loop  // branches if x1≠x2
```

If we predict that branches are taken, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without changing the code—in effect, the loop is unrolled dynamically by the hardware using the reservation stations obtained by renaming to act as additional registers.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point load/stores or operations have completed. Figure 3.14 shows reservation stations, register status tables, and load and store buffers at this point. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to 1.0, provided the multiplies could complete in four clock cycles. With a latency of six cycles, additional iterations will need to be processed before the steady state can be reached. This requires more reservation stations to hold instructions that are in execution. As we will see later in this chapter, when extended with multiple issue instructions, Tomasulo's approach can sustain more than one instruction per clock.

A load and a store can be done safely out of order, provided they access different addresses. If a load and a store access the same address, one of two things happens:

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station `r` empty	`if (RegisterStat[rs].Qi≠0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`if (RegisterStat[rt].Qi≠0)`
		`{RS[r].Qk ← RegisterStat[rt].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`

Load or store	Buffer `r` empty	`if (RegisterStat[rs].Qi≠0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`RS[r].A ← imm; RS[r].Busy ← yes;`

Load only		`RegisterStat[rt].Qi ← r;`

Store only		`if (RegisterStat[rt].Qi≠0)`
		`{RS[r].Qk ← RegisterStat[rs].Qi}`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`

Execute FP operation	`(RS[r].Qj = 0)` and `(RS[r].Qk = 0)`	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & `r` is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`

Write result
FP operation
or load
	Execution complete at `r` & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result;`
		`RegisterStat[x].Qi ← 0});`
		`∀x(if (RS[x].Qj=r)`
		`{RS[x].Vj ←`
		`result;RS[x].Qj ← 0});`
		`∀x(if (RS[x].Qk=r)`
		`{RS[x].Vk ←`
		`result;RS[x].Qk ← 0});`
		`RS[r].Busy ← no;`

Store	Execution complete at `r` & `RS[r].Qk = 0`	`Mem[RS[r].A] ← RS[r].Vk;` `RS[r].Busy ← no;`

Figure 3.13 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the source register numbers, `imm` is the sign-extended immediate field, and `r` is the reservation station or buffer that the instruction is assigned to. `RS` is the reservation station data structure. The value returned by an FP unit or by the load unit is called `result`. `RegisterStat` is the register status data structure (not the register file, which is `Regs[]`). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose values of Qj or Qk are the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that, to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because no concept of program order is maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step if there is a pending branch already in the pipeline. In Section 3.6, we will see how speculation support removes this restriction.

Instruction		**Instruction status**			
		From iteration	**Issue**	**Execute**	**Write result**
`fld`	`f0,0(x1)`	1	√	√	
`fmul.d`	`f4,f0,f2`	1	√		
`fsd`	`f4,0(x1)`	1	√		
`fld`	`f0,0(x1)`	2	√	√	
`fmul.d`	`f4,f0,f2`	2	√		
`fsd`	`f4,0(x1)`	2	√		

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	Yes	Load					`Regs[x1]` + `0`
Load2	Yes	Load					`Regs[x1]` − `8`
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		`Regs[f2]`	Load1		
Mult2	Yes	MUL		`Regs[f2]`	Load2		
Store1	Yes	Store	`Regs[x1]`			Mult1	
Store2	Yes	Store	`Regs[x1]` − `8`			Mult2	

	Register status							
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	**...**	**f30**
Qi	Load2		Mult2						

Figure 3.14 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

- The load is before the store in program order and interchanging them results in a WAR hazard.
- The store is before the load in program order and interchanging them results in a RAW hazard.

Similarly, interchanging two stores to the same address results in a WAW hazard.

Therefore, to determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load. Similarly, a store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address. We consider a method to eliminate this restriction in Section 3.9.

To detect such hazards, the processor must have computed the data memory address associated with any earlier memory operation. A simple, but not necessarily optimal, way to guarantee that the processor has all such addresses is to perform the effective address calculations in program order. (We really only need to keep the relative order between stores and other memory references; that is, loads can be reordered freely.)

Let's consider the situation of a load first. If we perform effective address calculation in program order, then when a load has completed effective address calculation, we can check whether there is an address conflict by examining the A field of all active store buffers. If the load address matches the address of any active entries in the store buffer, that load instruction is not sent to the load buffer until the conflicting store completes. (Some implementations bypass the value directly to the load from a pending store, reducing the delay for this RAW hazard.)

Stores operate similarly, except that the processor must check for conflicts in both the load buffers and the store buffers because conflicting stores cannot be reordered with respect to either a load or a store.

A dynamically scheduled pipeline can yield very high performance, provided branches are predicted accurately—an issue we addressed in the previous section. The major drawback of this approach is the complexity of the Tomasulo scheme, which requires a large amount of hardware. In particular, each reservation station must contain an associative buffer, which must run at high speed, as well as complex control logic. The performance can also be limited by the single CDB. Although additional CDBs can be added, each CDB must interact with each reservation station, and the associative tag-matching hardware would have to be duplicated at each station for each CDB. In the 1990s, only high-end processors could take advantage of dynamic scheduling (and its extension to speculation); however, recently even processors designed for PMDs are using these techniques, and processors for high-end desktops and small servers have hundreds of buffers to support dynamic scheduling.

In Tomasulo's scheme, two different techniques are combined: the renaming of the architectural registers to a larger set of registers and the buffering of source operands from the register file. Source operand buffering resolves WAR hazards that arise when the operand is available in the registers. As we will see later, it is also possible to eliminate WAR hazards by the renaming of a register together with the buffering of a result until no outstanding references to the earlier version of the register remain. This approach will be used when we discuss hardware speculation.

Tomasulo's scheme was unused for many years after the 360/91, but was widely adopted in multiple-issue processors starting in the 1990s for several reasons:

1. Although Tomasulo's algorithm was designed before caches, the presence of caches, with the inherently unpredictable delays, has become one of the major motivations for dynamic scheduling. Out-of-order execution allows the processors to continue executing instructions while awaiting the completion of a cache miss, thus hiding all or part of the cache miss penalty.
2. As processors became more aggressive in their issue capability and designers were concerned with the performance of difficult-to-schedule code (such as most nonnumeric code), techniques such as register renaming, dynamic scheduling, and speculation became more important.
3. It can achieve high performance without requiring the compiler to target code to a specific pipeline structure, a valuable property in the era of shrink-wrapped mass market software.

3.6 Hardware-Based Speculation

As we try to exploit more instruction-level parallelism, maintaining control dependences becomes an increasing burden. Branch prediction reduces the direct stalls attributable to branches, but for a processor executing multiple instructions per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism. A wide-issue processor may need to execute a branch every clock cycle to maintain maximum performance. Thus exploiting more parallelism requires that we overcome the limitation of control dependence.

Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if our guesses are correct. This mechanism represents a subtle, but important, extension over branch prediction with dynamic scheduling. In particular, with speculation, we fetch, issue, and *execute* instructions, as if our branch predictions are always correct; dynamic scheduling only fetches and issues such instructions. Of course, we need mechanisms to handle the situation where the speculation is incorrect. Appendix H discusses a variety of mechanisms for supporting speculation by the compiler. In this section, we explore *hardware speculation,* which extends the ideas of dynamic scheduling.

Hardware-based speculation combines three key ideas: (1) dynamic branch prediction to choose which instructions to execute, (2) speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and (3) dynamic scheduling to deal with the scheduling of different combinations of basic blocks. (In comparison, dynamic scheduling without speculation only partially overlaps basic blocks because it requires that a branch be resolved before actually executing any instructions in the successor basic block.)

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data flow execution*: Operations execute as soon as their operands are available.

To extend Tomasulo's algorithm to support speculation, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative.

Using the bypassed value is like performing a speculative register read because we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence *instruction commit*.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit *in order* and to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits. Therefore, when we add speculation, we need to separate the process of completing execution from instruction commit, because instructions may finish execution considerably before they are ready to commit. Adding this commit phase to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the *reorder buffer*, is also used to pass results among instructions that may be speculated.

The reorder buffer (ROB) provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set. The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits. The ROB therefore is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo's algorithm. The key difference is that in Tomasulo's algorithm, once an instruction writes its result, all subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the interval between completion of instruction execution and instruction commit. The ROB is similar to the store buffer in Tomasulo's algorithm, and we integrate the function of the store buffer into the ROB for simplicity.

Figure 3.15 shows the hardware structure of the processor including the ROB. Each entry in the ROB contains four fields: the instruction type, the destination field, the value field, and the ready field. The instruction type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which has register destinations). The destination field supplies the register number (for loads

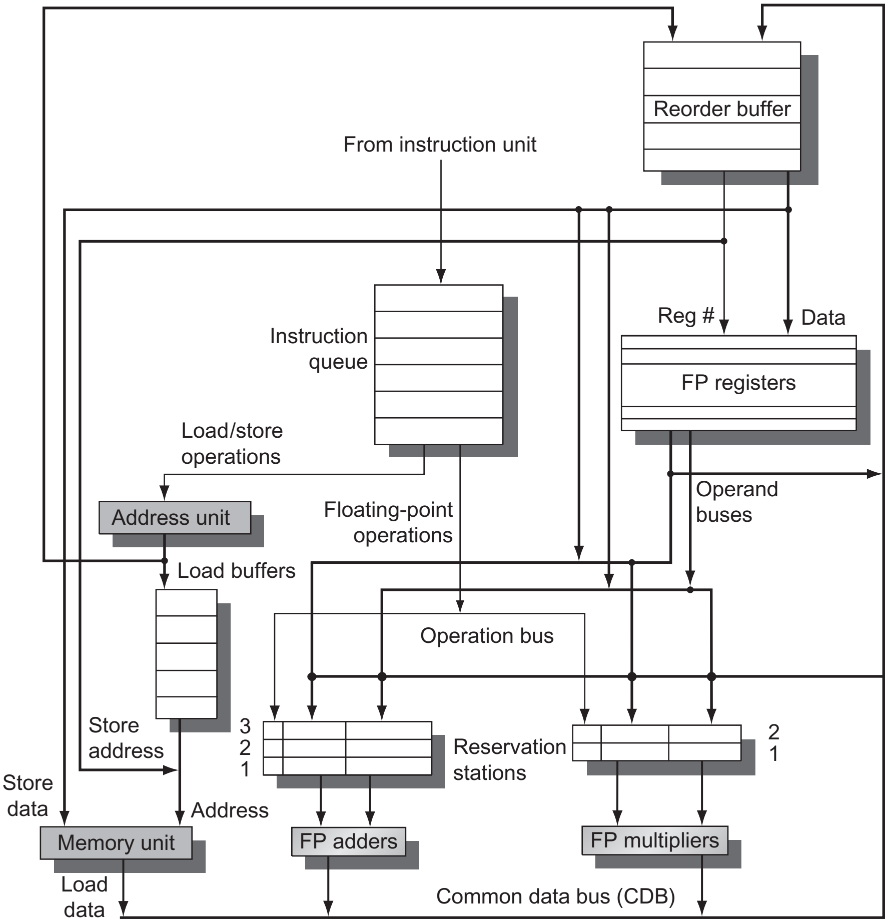

Reorder buffer
From instruction unit
Reg #
Data
Instruction queue
FP registers
Load/store operations
Operand buses
Address unit
Floating-point operations
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.10 on page 198, which implemented Tomasulo's algorithm, we can see that the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to allow multiple issues per clock by making the CDB wider to allow for multiple completions per clock.

and ALU operations) or the memory address (for stores) where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits. We will see an example of ROB entries shortly. Finally, the ready field indicates that the instruction has completed execution, and the value is ready.

The ROB subsumes the store buffers. Stores still execute in two steps, but the second step is performed by instruction commit. Although the renaming function of the reservation stations is replaced by the ROB, we still need a place to buffer operations (and operands) between the time they issue and the time they begin execution.

This function is still provided by the reservation stations. Because every instruction has a position in the ROB until it commits, we tag a result using the ROB entry number rather than using the reservation station number. This tagging requires that the ROB assigned for an instruction must be tracked in the reservation station. Later in this section, we will explore an alternative implementation that uses extra registers for renaming and a queue that replaces the ROB to decide when instructions can commit.

Here are the four steps involved in instruction execution:

1. *Issue*—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB; send the operands to the reservation station if they are available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry allocated for the result is also sent to the reservation station so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then the instruction issue is stalled until both have available entries.
2. *Execute*—If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores only need the base register at this step, because execution for a store at this point is only effective address calculation.
3. *Write result*—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity we assume that this occurs during the Write Result stage of a store; we discuss relaxing this requirement later.
4. *Commit*—This is the final stage of completing an instruction, after which only its result remains. (Some processors call this commit phase "completion" or "graduation.") There are three different sequences of actions at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished.

Once an instruction commits, its entry in the ROB is reclaimed, and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free. Now let's examine how this scheme would work with the same example we used for Tomasulo's algorithm.

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the following code segment, the same one we used to generate Figure 3.12, show what the status tables look like when the `fmul.d` is ready to go to commit.

```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6,f8,f2
```

Answer Figure 3.16 shows the result in the three tables. Notice that although the `fsub.d` instruction has completed execution, it does not commit until the `fmul.d` commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 200 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we added the Dest field to the reservation stations. The Dest field designates the ROB entry that is the destination for the result produced by this reservation station entry.

The preceding example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 3.16 with that of Figure 3.12 on page 184, which shows the same code sequence in operation on a processor with Tomasulo's algorithm. The key difference is that, in the preceding example, no instruction after the earliest uncompleted instruction (`fmul.d` in preceding example) is allowed to complete. In contrast, in Figure 3.12 the `fsub.d` and `fadd.d` instructions have also completed.

One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model. For example, if the `fmul.d` instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions from the ROB. Because instruction commit happens in order, this yields a precise exception.

By contrast, in the example using Tomasulo's algorithm, the `fsub.d` and `fadd.d` instructions could both complete before the `fmul.d` raised the exception. The result is that the registers `f8` and `f6` (destinations of the `fsub.d` and

	Reorder buffer					
Entry	**Busy**	**Instruction**		**State**	**Destination**	**Value**
1	No	`fld`	`f6,32(x2)`	Commit	`f6`	`Mem[32 + Regs[x2]]`
2	No	`fld`	`f2,44(x3)`	Commit	`f2`	`Mem[44 + Regs[x3]]`
3	Yes	`fmul.d`	`f0,f2,f4`	Write result	`f0`	#2 × `Regs[f4]`
4	Yes	`fsub.d`	`f8,f2,f6`	Write result	`f8`	#2 − #1
5	Yes	`fdiv.d`	`f0,f0,f6`	Execute	`f0`	
6	Yes	`fadd.d`	`f6,f8,f2`	Write result	`f6`	#4 + #2

	Reservation stations							
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**Dest**	**A**
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	`fmul.d`	`Mem[44 + Regs[x3]]`	`Regs[f4]`			#3	
Mult2	Yes	`fdiv.d`		`Mem[32 + Regs[x2]]`	#3		#5	

	FP register status								
Field	**f0**	**f1**	**f2**	**f3**	**f4**	**f5**	**f6**	**f7**	**f8**	**f10**
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

Figure 3.16 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are there only to ease understanding. The `fsub.d` and `fadd.d` instructions will not commit until the `fmul.d` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `fdiv.d` is in execution, but has not completed solely because of its longer latency than that of `fmul.d`. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed but are shown for informational purposes. We do not show the entries for the load/store queue, but these entries are kept in order.

`fadd.d` instructions) could be overwritten, in which case the interrupt would be imprecise.

Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors because the program will likely terminate; see Appendix J for further discussion of this topic. Other types of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise because the program must transparently resume execution after handling such an exception.

The use of a ROB with in-order instruction commit provides precise exceptions, in addition to supporting speculative execution, as the next example shows.

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 3.14 in execution:

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,−8
       bne     x1,x2,Loop    //branches if x1≠x2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the `fld` and `fmul.d` from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (`x1` in this example) and the value (`f4` in this example). Because we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer Figure 3.17 shows the result in two tables.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that the branch `bne` is not taken the first time in Figure 3.17. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.

In practice, processors that speculate try to recover as early as possible after a branch is mispredicted. This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch, allowing those that are before the branch in the ROB to continue, and restarting the fetch at the correct branch successor. In speculative processors, performance is more sensitive to the branch prediction because the impact of a misprediction will be higher. Thus all the aspects of handling branches—prediction accuracy, latency of misprediction detection, and misprediction recovery time—increase in importance.

Exceptions are handled by not recognizing the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the

	Reorder buffer					
Entry	**Busy**	**Instruction**		**State**	**Destination**	**Value**
1	No	`fld`	`f0,0(x1)`	Commit	`f0`	`Mem[0 + Regs[x1]]`
2	No	`fmul.d`	`f4,f0,f2`	Commit	`f4`	#1 × `Regs[f2]`
3	Yes	`fsd`	`f4,0(x1)`	Write result	`0 + Regs[x1]`	#2
4	Yes	`addi`	`x1,x1,–8`	Write result	`x1`	`Regs[x1] – 8`
5	Yes	`bne`	`x1,x2,Loop`	Write result		
6	Yes	`fld`	`f0,0(x1)`	Write result	`f0`	`Mem[#4]`
7	Yes	`fmul.d`	`f4,f0,f2`	Write result	`f4`	#6 × `Regs[f2]`
8	Yes	`fsd`	`f4,0(x1)`	Write result	0 + #4	#7
9	Yes	`addi`	`x1,x1,–8`	Write result	`x1`	#4 – 8
10	Yes	`bne`	`x1,x2,Loop`	Write result		

	FP register status								
Field	**f0**	**f1**	**f2**	**f3**	**f4**	**F5**	**f6**	**F7**	**f8**
Reorder #	6								
Busy	Yes	No	No	No	Yes	No	No	…	No

Figure 3.17 Only the `fld` and `fmul.d` instructions have committed, although all the others have completed execution. Thus no reservation stations are busy and none are shown. The remaining instructions will be committed as quickly as possible. The first two reorder buffers are empty, but are shown for completeness.

ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. We can also try to handle exceptions as soon as they arise and all earlier branches are resolved, but this is more challenging in the case of exceptions than for branch mispredict and, because it occurs less frequently, not as critical.

Figure 3.18 shows the steps of execution for an instruction, as well as the conditions that must be satisfied to proceed to the step and the actions taken. We show the case where mispredicted branches are not resolved until commit. Although speculation seems like a simple addition to dynamic scheduling, a comparison of Figure 3.18 with the comparable figure for Tomasulo's algorithm in Figure 3.13 shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex.

There is an important difference in how stores are handled in a speculative processor versus in Tomasulo's algorithm. In Tomasulo's algorithm, a store can update memory when it reaches Write Result (which ensures that the effective address has been calculated) and the data value to store is available. In a speculative processor, a store updates memory only when it reaches the head of

Status | **Wait until** | **Action or bookkeeping**

Issue all instructions | Reservation station (r) and ROB (b) both available

```
if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/
   {h ← RegisterStat[rs].Reorder;
   if (ROB[h].Ready)/* Instr completed already */
      {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;}
   else {RS[r].Qj ← h;} /* wait for instruction */
} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;};
RS[r].Busy ← yes; RS[r].Dest ← b;
ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;
```

FP operations and stores

```
if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/
   {h ← RegisterStat[rt].Reorder;
   if (ROB[h].Ready)/* Instr completed already */
      {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;}
   else {RS[r].Qk ← h;} /* wait for instruction */
} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};
```

FP operations

```
RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes;
ROB[b].Dest ← rd;
```

Loads

```
RS[r].A ← imm; RegisterStat[rt].Reorder ← b;
RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;
```

Stores

```
RS[r].A ← imm;
```

Execute FP op | (RS[r].Qj == 0) and (RS[r].Qk == 0)

Compute results—operands are in Vj and Vk

Load step 1 | (RS[r].Qj == 0) and there are no stores earlier in the queue

```
RS[r].A ← RS[r].Vj + RS[r].A;
```

Load step 2 | Load step 1 done and all stores earlier in ROB have different address

Read from `Mem[RS[r].A]`

Store | (RS[r].Qj == 0) and store at queue head

```
ROB[h].Address ← RS[r].Vj + RS[r].A;
```

Write result all but store | Execution done at *r* and CDB available

```
b ← RS[r].Dest; RS[r].Busy ← no;
∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0});
∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0});
ROB[b].Value ← result; ROB[b].Ready ← yes;
```

Store | Execution done at *r* and (RS[r].Qk == 0)

```
ROB[h].Value ← RS[r].Vk;
```

Commit | Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes

```
d ← ROB[h].Dest; /* register dest, if exists */
if (ROB[h].Instruction==Branch)
   {if (branch is mispredicted)
    {clear ROB[h], RegisterStat; fetch branch dest;};}
else if (ROB[h].Instruction==Store)
        {Mem[ROB[h].Destination] ← ROB[h].Value;}
else /* put the result in the register destination */
   {Regs[d] ← ROB[h].Value;};
ROB[h].Busy ← no; /* free up ROB entry */
/* free up dest register if no one else writing it */
if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};
```

Figure 3.18 Steps in the algorithm and what is required for each step. For the issuing instruction, `rd` is the destination, `rs` and `rt` are the sources, `r` is the reservation station allocated, `b` is the assigned ROB entry, and `h` is the head entry of the ROB. `RS` is the reservation station data structure. The value returned by a reservation station is called the `result`. `Register-Stat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 3.18 has one significant simplification for stores, which is unneeded in practice. Figure 3.18 requires stores to wait in the Write Result stage for the register source operand whose value is to be stored; the value is then moved from the Vk field of the store's reservation station to the Value field of the store's ROB entry. In reality, however, the value to be stored need not arrive until *just before* the store commits and can be placed directly into the store's ROB entry by the sourcing instruction. This is accomplished by having the hardware track when the source value to be stored is available in the store's ROB entry and searching the ROB on every instruction completion to look for dependent stores.

This addition is not complicated, but adding it has two effects: we would need to add a field to the ROB, and Figure 3.18, which is already in a small font, would be even longer! Although Figure 3.18 makes this simplification, in our examples, we will allow the store to pass through the Write Result stage and simply wait for the value to be ready when it commits.

Like Tomasulo's algorithm, we must avoid hazards through memory. WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB, so no earlier loads or stores can still be pending. RAW hazards through memory are maintained by two restrictions:

1. Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load
2. Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data. Some speculative processors will actually bypass the value from the store to the load directly when such a RAW hazard occurs. Another approach is to predict potential collisions using a form of value prediction; we consider this in Section 3.9.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units. Indeed, because such programs tend to have code where the branch behavior is less predictable, speculation may be more useful in integer programs. Additionally, these techniques can be extended to work in a multiple-issue processor by allowing multiple instructions to issue and commit every clock. In fact, speculation is probably most interesting in such processors because less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

3.7 Exploiting ILP Using Multiple Issue and Static Scheduling

The techniques of the preceding sections can be used to eliminate data, control stalls, and achieve an ideal CPI of one. To improve performance further, we want to decrease the CPI to less than one, but the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

The goal of the *multiple-issue processors*, discussed in the next few sections, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three major flavors:

1. Statically scheduled superscalar processors
2. VLIW (very long instruction word) processors
3. Dynamically scheduled superscalar processors

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction. VLIW processors are inherently statically scheduled by the compiler. When Intel and HP created the IA-64 architecture, described in Appendix H, they also introduced the name EPIC (explicitly parallel instruction computer) for this architectural style.

Although statically scheduled superscalars issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs because both approaches rely on the compiler to schedule code for the processor. Because of the diminishing advantages of a statically scheduled superscalar as the issue width grows, statically scheduled superscalars are used primarily for narrow issue widths, normally just two instructions. Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled superscalar. Because of the similarities in hardware and required compiler technology, we focus on VLIWs in this section, and we will see them again in Chapter 7. The insights of this section are easily extrapolated to a statically scheduled superscalar.

Figure 3.19 summarizes the basic approaches to multiple issue and their distinguishing characteristics and shows processors that use each approach.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction or requires that the instructions in the

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

issue packet satisfy the same constraints. Because there is no fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach.

Because the advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider issue processor. Indeed, for simple two-issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four-issue processor has manageable overhead, but as we will see later in this chapter, the growth in overhead is a major factor limiting wider issue processors.

Let's consider a VLIW processor with instructions that contain five operations, including one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain six operations per instruction packet (i.e., they allow concurrent issue of two three-instruction bundles, as Appendix H describes).

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straight-line code, then *local scheduling* techniques, which operate on a single basic block, can be used. If finding and exploiting the parallelism require scheduling code across branches, a substantially more complex *global* *scheduling* algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they also must deal with significantly more complicated trade-offs in optimization, because moving code across branches is expensive.

In Appendix H, we discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs; we will also explore special hardware support that allows some conditional branches to be eliminated, extending the usefulness of local scheduling and enhancing the performance of global scheduling.

For now, we will rely on loop unrolling to generate long, straight-line code sequences so that we can use local scheduling to build up VLIW instructions and focus on how well these processors operate.

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop `x[i]=x[i]+s` (see page 158 for the RISC-V code) for such a processor. Unroll as many times as necessary to eliminate any stalls.

Answer Figure 3.20 shows the code. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles for the unrolled and scheduled loop. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 3.2 that used unrolled and scheduled code.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
`fld f0,0(x1)`	`fld f6,-8(x1)`			
`fld f10,-16(x1)`	`fld f14,-24(x1)`			
`fld f18,-32(x1)`	`fld f22,-40(x1)`	`fadd.d f4,f0,f2`	`fadd.d f8,f6,f2`	
`fld f26,-48(x1)`		`fadd.d f12,f0,f2`	`fadd.d f16,f14,f2`	
		`fadd.d f20,f18,f2`	`fadd.d f24,f22,f2`	
`fsd f4,0(x1)`	`fsd f8,-8(x1)`	`fadd.d f28,f26,f24`		
`fsd f12,-16(x1)`	`fsd f16,-24(x1)`			`addi x1,x1,-56`
`fsd f20,24(x1)`	`fsd f24,16(x1)`			
`fsd f28,8(x1)`				`bne x1,x2,Loop`

Figure 3.20 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming correct branch prediction. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than RISC-V would normally use in this loop. The preceding VLIW code sequence requires at least eight FP registers, whereas the same code sequence for the base RISC-V processor can use as few as two FP registers or as many as five when unrolled and scheduled.

For the original VLIW model, there were both technical and logistical problems that made the approach less efficient. The technical problems were the increase in code size and the limitations of the lockstep operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as in earlier examples), thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Appendix H, we examine software scheduling approaches, such as software pipelining, that can achieve the benefits of unrolling without as much code expansion.

To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded. In Appendix H, we show other techniques, as well as document the significant code expansion seen in IA-64.

Early VLIWs operated in lockstep; there was no hazard-detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall because all the functional units had to be kept synchronized. Although a compiler might have been able to schedule the deterministic functional units to prevent stalls, predicting which data accesses would encounter a cache stall and scheduling them were very difficult to do. Thus caches needed to be blocking and causing *all* the functional units to stall. As the issue rate and number of memory references became large, this synchronization restriction became unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for general-purpose VLIWs or those that run third-party software. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach. In the domain-specific architectures, which we examine in Chapter 7, this problem is not serious because applications are written specifically for an architectural configuration.

The EPIC approach, of which the IA-64 architecture is the primary example, provides solutions to many of the problems encountered in early general-purpose VLIW designs, including extensions for more aggressive software speculation and methods to overcome the limitation of hardware dependence while preserving binary compatibility.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP programs, the original loop probably could have been run efficiently on a vector processor (described in the next chapter). It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are the former's ability to extract some parallelism from less structured code and to easily cache all forms of data. For these reasons, multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

3.8 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

So far we have seen how the individual mechanisms of dynamic scheduling, multiple issue, and speculation work. In this section, we put all three together, which yields a microarchitecture quite similar to those in modern microprocessors. For simplicity we consider only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

Let's assume we want to extend Tomasulo's algorithm to support multiple-issue superscalar pipeline with separate integer, load/store, and floating-point units (both FP multiply and FP add), each of which can initiate an operation on every clock. We do not want to issue instructions to the reservation stations out of order because this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling, we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit. Because the interaction of the integer and floating-point instructions is crucial, we also extend Tomasulo's scheme to deal with both the integer and floating-point functional units and registers, as well as incorporating speculative execution. As Figure 3.21 shows, the basic organization is similar to that of a processor with speculation with one issue per clock, except that the issue and completion logic must be enhanced to allow multiple instructions to be processed per clock.

Issuing multiple instructions per clock in a dynamically scheduled processor (with or without speculation) is very complex for the simple reason that the multiple instructions may depend on one another. Because of this, the tables must be updated for the instructions in parallel; otherwise, the tables will be incorrect or the dependence may be lost.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that the key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle so that two instructions can be processed in one clock cycle; this approach cannot be easily extended to handle four instructions per clock, unfortunately.

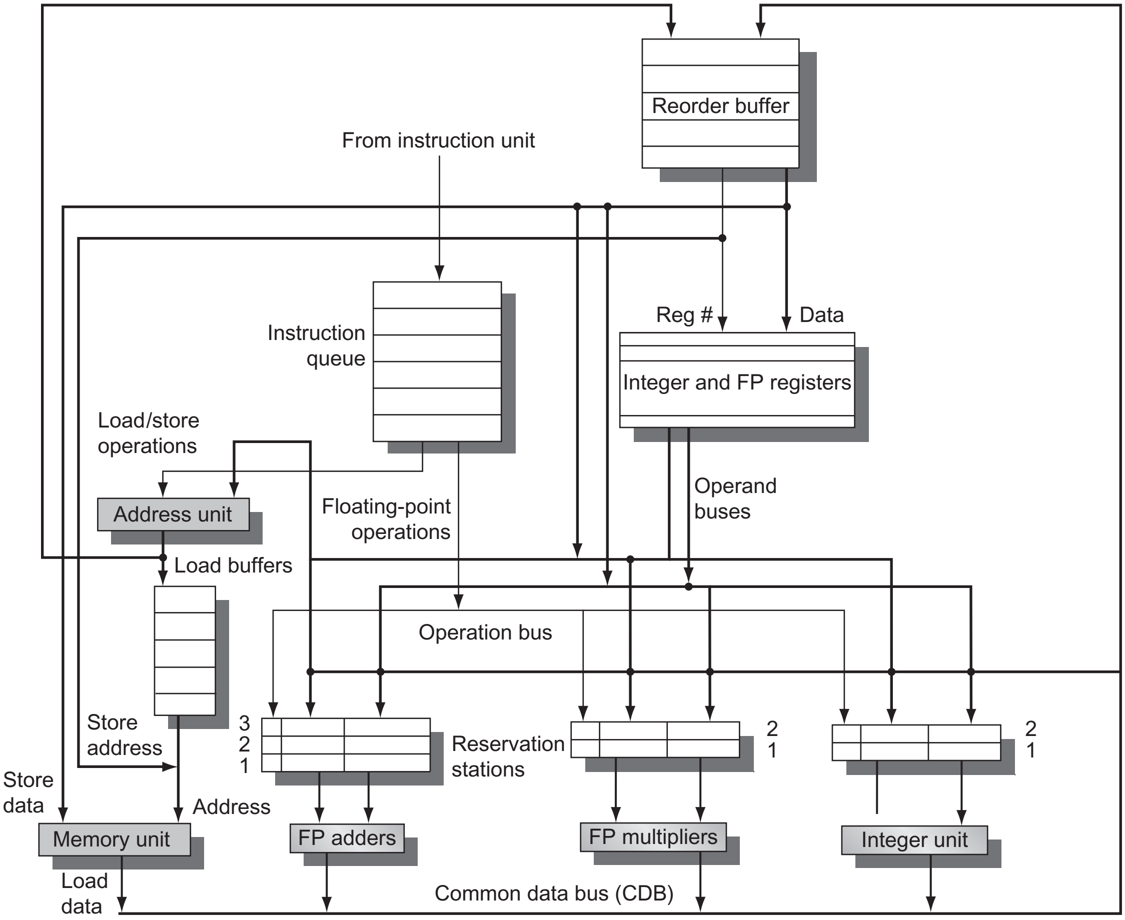

Reorder buffer
From instruction unit
Reg #
Data
Instruction queue
Integer and FP registers
Load/store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Integer unit
Load data
Common data bus (CDB)

Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

A second alternative is to build the logic necessary to handle two or more instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock may include both approaches: They both pipeline and widen the issue logic. A key observation is that we cannot simply pipeline away the problem. By making instruction issues take multiple clocks because new instructions are issuing every clock cycle, we must be able to assign the reservation station and to update the pipeline tables so that a dependent instruction issuing on the next clock can use the updated information.

This issue step is one of the most fundamental bottlenecks in dynamically scheduled superscalars. To illustrate the complexity of this process, Figure 3.22 shows the issue logic for one case: issuing a load followed by a dependent FP operation. The logic is based on that in Figure 3.18 on page 197, but represents only one case. In a modern superscalar, every possible combination of dependent instructions that is allowed to issue in the same clock cycle must be considered. Because the number of possibilities climbs as the square of the number of instructions that can be issued in a clock, the issue step is a likely bottleneck for attempts to go beyond four instructions per clock.

We can generalize the detail of Figure 3.22 to describe the basic strategy for updating the issue logic and the reservation tables in a dynamically scheduled superscalar with up to *n* issues per clock as follows:

1. Assign a reservation station and a reorder buffer for *every* instruction that *might* be issued in the next issue bundle. This assignment can be done before the instruction types are known simply by preallocating the reorder buffer entries sequentially to the instructions in the packet using *n* available reorder buffer entries and by ensuring that enough reservation stations are available to issue the whole bundle, independent of what it contains. By limiting the number of instructions of a given class (say, one FP, one integer, one load, one store), the necessary reservation stations can be preallocated. Should sufficient reservation stations not be available (such as when the next few instructions in the program are all of one instruction type), the bundle is broken, and only a subset of the instructions, in the original program order, is issued. The remainder of the instructions in the bundle can be placed in the next bundle for potential issue.
2. Analyze all the dependences among the instructions in the issue bundle.
3. If an instruction in the bundle depends on an earlier instruction in the bundle, use the assigned reorder buffer number to update the reservation table for the dependent instruction. Otherwise, use the existing reservation table and reorder buffer information to update the reservation table entries for the issuing instruction.

Of course, what makes the preceding very complicated is that it is all done in parallel in a single clock cycle!

At the back-end of the pipeline, we must be able to complete and commit multiple instructions per clock. These steps are somewhat easier than the issue problems because multiple instructions that can actually commit in the same clock cycle must have already dealt with and resolved any dependences. As we will see, designers have figured out how to handle this complexity: The Intel i7, which we examine in Section 3.12, uses essentially the scheme we have described for speculative multiple issue, including a large number of reservation stations, a reorder buffer, and a load and store buffer that is also used to handle nonblocking cache misses.

From a performance viewpoint, we can show how the concepts fit together with an example.

Action or bookkeeping — **Comments**

```
if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/
   {h ← RegisterStat[rs1].Reorder;
   if (ROB[h].Ready)/* Instr completed already */
       {RS[x1].Vj ← ROB[h].Value; RS[x1].Qj ← 0;}
   else {RS[x1].Qj ← h;} /* wait for instruction */
} else {RS[x1].Vj ← Regs[rs]; RS[x1].Qj ← 0;};
RS[x1].Busy ← yes; RS[x1].Dest ← b1;
ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1;
ROB[b1].Ready ← no;
RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1;
RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;
```

Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.

```
RS[x2].Qj ← b1;} /* wait for load instruction */
```

Because we know that the first operand of the FP operation is from the load, this step simply updates the reservation station to point to the load. Notice that the dependence must be analyzed on the fly and the ROB entries must be allocated during this issue step so that the reservation tables can be correctly updated.

```
if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/
   {h ← RegisterStat[rt2].Reorder;
   if (ROB[h].Ready)/* Instr completed already */
       {RS[x2].Vk ← ROB[h].Value; RS[x2].Qk ← 0;}
   else {RS[x2].Qk ← h;} /* wait for instruction */
} else {RS[x2].Vk ← Regs[rt2]; RS[x2].Qk ← 0;};
RegisterStat[rd2].Reorder ← b2;
RegisterStat[rd2].Busy ← yes;
ROB[b2].Dest ← rd2;
```

Because we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction were dependent on something in the same issue bundle, the tables would need to be updated using the assigned reservation buffer.

```
RS[x2].Busy ← yes; RS[x2].Dest ← b2;
ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2;
ROB[b2].Ready ← no;
```

This section simply updates the tables for the FP operation and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.

Figure 3.22 The issue steps for a pair of dependent instructions (called 1 and 2), where instruction 1 is FP load and instruction 2 is an FP operation whose first operand is the result of the load instruction; `x1` and `x2` are the assigned reservation stations for the instructions; and `b1` and `b2` are the assigned reorder buffer entries. For the issuing instructions, `rd1` and `rd2` are the destinations; `rs1`, `rs2`, and `rt2` are the sources (the load has only one source); `x1` and `x2` are the reservation stations allocated; and `b1` and `b2` `are` the assigned ROB entries. `RS` is the reservation station data structure. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure. Notice that we need to have assigned reorder buffer entries for this logic to operate properly, and recall that all these updates happen in a single clock cycle in parallel, not sequentially.

Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:  ld    x2,0(x1)       //x2=array element
       addi  x2,x2,1        //increment x2
       sd    x2,0(x1)       //store result
       addi  x1,x1,8        //increment pointer
       bne   x2,x3,Loop     //branch if not last
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.23 and 3.24 show the performance for a two-issue, dynamically scheduled processor, without and with speculation. In this case, where a branch

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	`ld x2,0(x1)`	1	2	3	4	First issue
1	`addi x2,x2,1`	1	5		6	Wait for `ld`
1	`sd x2,0(x1)`	2	3	7		Wait for `addi`
1	`addi x1,x1,8`	2	3		4	Execute directly
1	`bne x2,x3,Loop`	3	7			Wait for `addi`
2	`ld x2,0(x1)`	4	8	9	10	Wait for `bne`
2	`addi x2,x2,1`	4	11		12	Wait for `ld`
2	`sd x2,0(x1)`	5	9	13		Wait for `addi`
2	`addi x1,x1,8`	5	8		9	Wait for `bne`
2	`bne x2,x3,Loop`	6	13			Wait for `addi`
3	`ld x2,0(x1)`	7	14	15	16	Wait for `bne`
3	`addi x2,x2,1`	7	17		18	Wait for `ld`
3	`sd x2,0(x1)`	8	15	19		Wait for `addi`
3	`addi x1,x1,8`	8	14		15	Wait for `bne`
3	`bne x2,x3,Loop`	9	19			Wait for `addi`

Figure 3.23 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the `ld` following the `bne` cannot start execution earlier because it must wait until the branch outcome is determined. This type of program, with data-dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch-condition evaluation allow multiple instructions to execute in the same cycle. Figure 3.24 shows this example with speculation.

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	`ld x2,0(x1)`	1	2	3	4	5	First issue
1	`addi x2,x2,1`	1	5		6	7	Wait for `ld`
1	`sd x2,0(x1)`	2	3			7	Wait for `addi`
1	`addi x1,x1,8`	2	3		4	8	Commit in order
1	`bne x2,x3,Loop`	3	7			8	Wait for `addi`
2	`ld x2,0(x1)`	4	5	6	7	9	No execute delay
2	`addi x2,x2,1`	4	8		9	10	Wait for `ld`
2	`sd x2,0(x1)`	5	6			10	Wait for `addi`
2	`addi x1,x1,8`	5	6		7	11	Commit in order
2	`bne x2,x3,Loop`	6	10			11	Wait for `addi`
3	`ld x2,0(x1)`	7	8	9	10	12	Earliest possible
3	`addi x2,x2,1`	7	11		12	13	Wait for `ld`
3	`sd x2,0(x1)`	8	9			13	Wait for `addi`
3	`addi x1,x1,8`	8	9		10	14	Executes earlier
3	`bne x2,x3,Loop`	9	13			14	Wait for `addi`

Figure 3.24 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the `ld` following the `bne` can start execution early because it is speculative.

can be a critical performance limiter, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 13, whereas it executes in clock cycle 19 on the nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to complete effective address calculation before a branch is decided, but unless speculative memory accesses are allowed, this improvement will gain only 1 clock per iteration.

This example clearly shows how speculation can be advantageous when there are data-dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. Incorrect speculation does not improve performance; in fact, it typically harms performance and, as we shall see, dramatically lowers energy efficiency.

3.9 Advanced Techniques for Instruction Delivery and Speculation

In a high-performance pipeline, especially one with multiple issues, predicting branches well is not enough; we actually have to be able to deliver a high-bandwidth instruction stream. In recent multiple-issue processors, this has meant delivering 4–8 instructions every clock cycle. We look at methods for increasing instruction delivery bandwidth first. We then turn to a set of key issues in implementing advanced speculation techniques, including the use of register renaming versus reorder buffers, the aggressiveness of speculation, and a technique called *value prediction*, which attempts to predict the result of a computation and which could further enhance ILP.

Increasing Instruction Fetch Bandwidth

A multiple-issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput. Of course, fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches. In this section, we look at two methods for dealing with branches and then discuss how modern processors integrate the instruction prediction and prefetch functions.

Branch-Target Buffers

To reduce the branch penalty for our simple five-stage pipeline, as well as for deeper pipelines, we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next program counter (PC) should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a *branch-target buffer* or *branch-target cache.* Figure 3.25 shows a branch-target buffer.

Because a branch-target buffer predicts the next instruction address and will send it out *before* decoding the instruction, we *must* know whether the fetched instruction is predicted as a taken branch. If the PC of the fetched instruction matches an address in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that unlike a branch-prediction buffer, the predictive entry must be matched to this instruction because the predicted PC will be sent

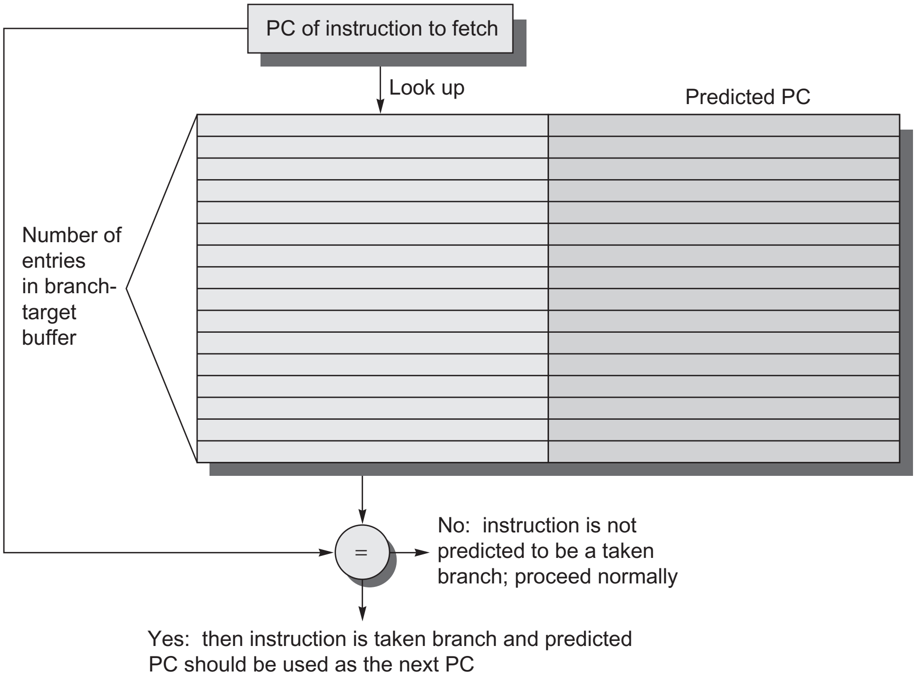

PC of instruction to fetch
Look up
Predicted PC
Number of entries in branch-target buffer
=
No: instruction is not predicted to be a taken branch; proceed normally
Yes: then instruction is taken branch and predicted PC should be used as the next PC

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

out before it is known whether this instruction is even a branch. If the processor did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in worse performance. We need to store only the predicted-taken branches in the branch-target buffer because an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

Figure 3.26 shows the steps when using a branch-target buffer for a simple five-stage pipeline. As we can see in this figure, there will be no branch delay if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least two clock cycles. Dealing with the mispredictions and misses is a significant challenge because we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus we want to make this process fast to minimize the penalty.

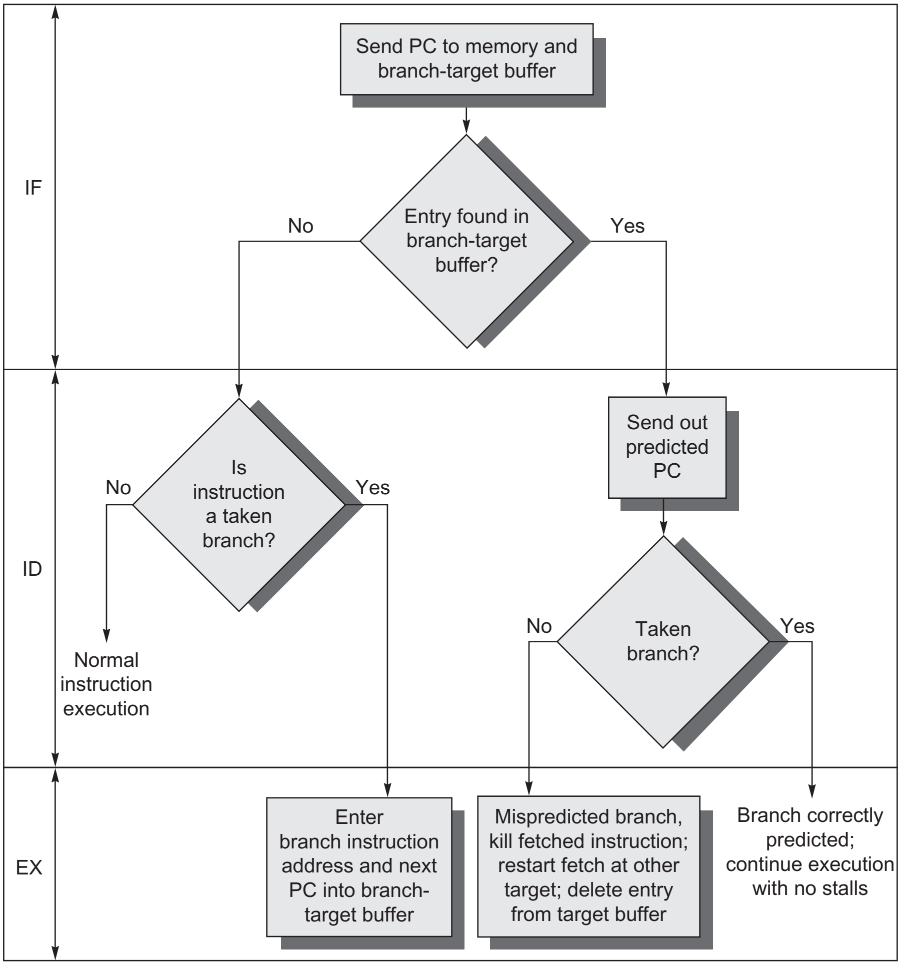

IF
ID
EX
Send PC to memory and branch-target buffer
Entry found in branch-target buffer?
No
Yes
Is instruction a taken branch?
No
Yes
Normal instruction execution
Send out predicted PC
Taken branch?
No
Yes
Enter branch instruction address and next PC into branch-target buffer
Mispredicted branch, kill fetched instruction; restart fetch at other target; delete entry from target buffer
Branch correctly predicted; continue execution with no stalls

Figure 3.26 The steps involved in handling an instruction with a branch-target buffer.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 3.27 contains this information for a simple five-stage pipeline.

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions in Figure 3.27. Make the following assumptions about the prediction accuracy and hit rate:

- Prediction accuracy is 90% (for instructions in the buffer).
- Hit rate in the buffer is 90% (for branches predicted taken).

Answer We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.

$$\begin{aligned}\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate}\\ &\quad \times \text{Percent incorrect predictions}\\ &= 90\% \times 10\% = 0.09\\ \text{Probability (branch not in buffer, but actually taken)} &= 10\%\\ \text{Branch penalty} &= (0.09 + 0.10) \times 2\\ \text{Branch penalty} &= 0.38\end{aligned}$$

The improvement from dynamic branch prediction will grow as the pipeline length, and thus the branch delay grows; in addition, better predictors will yield a greater performance advantage. Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

One variation on the branch-target buffer is to store one or more *target instructions* instead of, or in addition to, the predicted *target address*. This variation has two potential advantages. First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer. Second, buffering the actual target instructions allows us to perform an optimization called *branch folding*. Branch folding can be used to obtain 0-cycle unconditional branches and sometimes 0-cycle conditional branches. As we will see, the Cortex A-53 uses a single-entry branch target cache that stores the predicted target instructions.

Consider a branch-target buffer that buffers instructions from the predicted path and is being accessed with the address of an unconditional branch. The only function of the unconditional branch is to change the PC. Thus, when the branch-target buffer signals a hit and indicates that the branch is

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

Figure 3.27 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

unconditional, the pipeline can simply substitute the instruction from the branch-target buffer in place of the instruction that is returned from the cache (which is the unconditional branch). If the processor is issuing multiple instructions per cycle, then the buffer will need to supply multiple instructions to obtain the maximum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

As we try to increase the opportunity and accuracy of speculation, we face the challenge of predicting indirect jumps, that is, jumps whose destination address varies at runtime. High-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, although many indirect jumps simply come from procedure returns. For example, for the SPEC95 benchmarks, procedure returns account for more than 15% of the branches and the vast majority of the indirect jumps on average. For object-oriented languages such as C++ and Java, procedure returns are even more frequent. Thus focusing on procedure returns seems appropriate.

Though procedure returns can be predicted with a branch-target buffer, the accuracy of such a prediction technique can be low if the procedure is called from multiple sites and the calls from one site are not clustered in time. For example, in SPEC CPU95, an aggressive branch predictor achieves an accuracy of less than 60% for such return branches. To overcome this problem, some designs use a small buffer of return addresses operating as a stack. This structure caches the most recent return addresses, pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large (i.e., as large as the maximum call depth), it will predict the returns perfectly. Figure 3.28 shows the performance of such a return buffer with 0–16 elements for a number of the SPEC CPU95 benchmarks. We will use a similar return predictor when we examine the studies of ILP in Section 3.10. Both the Intel Core processors and the AMD Phenom processors have return address predictors.

In large server applications, indirect jumps also occur for various function calls and control transfers. Predicting the targets of such branches is not as simple as in a procedure return. Some processors have opted to add specialized predictors for all indirect jumps, whereas others rely on a branch target buffer.

Although a simple predictor like gshare does a good job of predicting many conditional branches, it is not tailored to predicting loop branches, especially for long running loops. As we observed earlier, the Intel Core i7 920 used a specialized loop branch predictor. With the emergence of tagged hybrid predictors, which are as good at predicting loop branches, some recent designers have opted to put the resources into larger tagged hybrid predictors rather than a separate loop branch predictor.

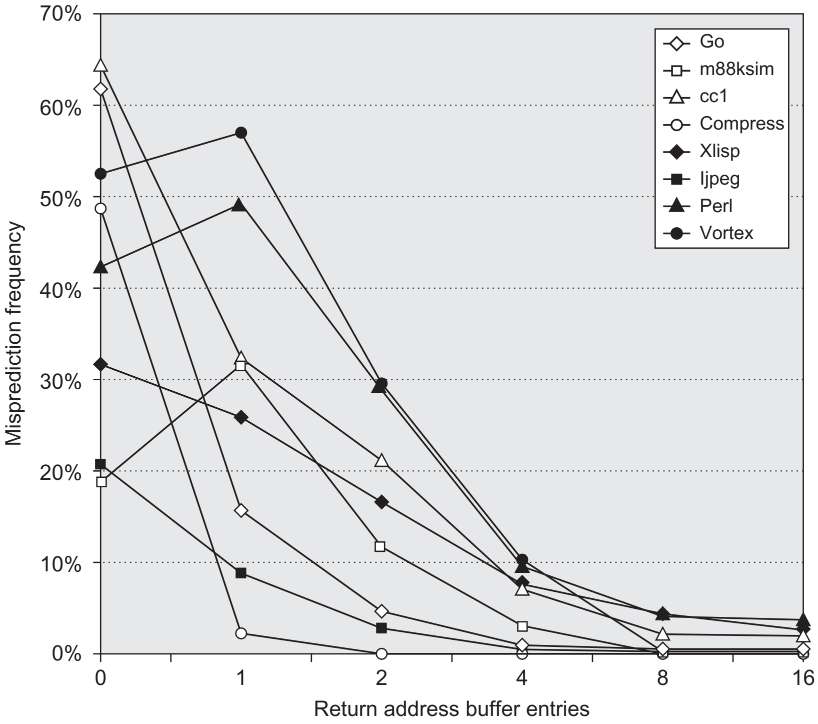


Figure 3.28 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. (1999) and use a fix-up mechanism to prevent corruption of the cached return addresses.

Integrated Instruction Fetch Units

To meet the demands of multiple-issue processors, many recent designers have chosen to implement an integrated instruction fetch unit as a separate autonomous unit that feeds instructions to the rest of the pipeline. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipe stage given the complexities of multiple issue is no longer valid.

Instead, recent designs have used an integrated instruction fetch unit that integrates several functions:

1. *Integrated branch prediction*—The branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so as to drive the fetch pipeline.

2. *Instruction prefetch*—To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions (see Chapter 2 for a discussion of techniques for doing this), integrating it with branch prediction.
3. *Instruction memory access and buffering*—When fetching multiple instructions per cycle, a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

Virtually all high-end processors now use a separate instruction fetch unit connected to the rest of the pipeline by a buffer containing pending instructions.

Speculation: Implementation Issues and Extensions

In this section, we explore five issues that involve the design trade-offs and challenges in multiple-issue and speculation, starting with the use of register renaming, the approach that is sometimes used instead of a reorder buffer. We then discuss one important possible extension to speculation on control flow: an idea called value prediction.

Speculation Support: Register Renaming Versus Reorder Buffers

One alternative to the use of a reorder buffer (ROB) is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo's algorithm and extends it. In Tomasulo's algorithm, the values of the *architecturally visible registers* (x0, . . . r31 and f0, . . . f31) are contained, at any point in execution, in some combination of the register set and the reservation stations. With the addition of speculation, register values may also temporarily reside in the ROB. In either case, if the processor does not issue new instructions for a period of time, all existing instructions will commit, and the register values will appear in the register file, which directly corresponds to the architecturally visible registers.

In the register-renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus the extended registers replace most of the function of the ROB and the reservation stations; only a queue to ensure that instructions complete in order is needed. During instruction issue, a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits.

The *renaming map* is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register, a function performed by the register status table in Tomasulo's algorithm. When an instruction commits, the renaming table is permanently updated to indicate that a physical register corresponds to the actual architectural register, thus effectively finalizing the update to the processor state. Although an ROB is not necessary with register renaming, the hardware must still track instructions in a queue-like structure and update the renaming table in strict order.

An advantage of the renaming approach versus the ROB approach is that instruction commit is slightly simplified because it requires only two simple actions: (1) record that the mapping between an architectural register number and physical register number is no longer speculative, and (2) free up any physical registers being used to hold the "older" value of the architectural register. In a design with reservation stations, a station is freed up when the instruction using it completes execution, and a ROB entry is freed up when the corresponding instruction commits.

With register renaming, deallocating registers is more complex because before we free up a physical register, we must know that it no longer corresponds to an architectural register and that no further uses of the physical register are outstanding. A physical register corresponds to an architectural register until the architectural register is rewritten, causing the renaming table to point elsewhere. That is, if no renaming entry points to a particular physical register, then it no longer corresponds to an architectural register. There may, however, still be outstanding uses of the physical register. The processor can determine whether this is the case by examining the source register specifiers of all instructions in the functional unit queues. If a given physical register does not appear as a source and it is not designated as an architectural register, it may be reclaimed and reallocated.

Alternatively, the processor can simply wait until another instruction that writes the same architectural register commits. At that point, there can be no further uses of the older value outstanding. Although this method may tie up a physical register slightly longer than necessary, it is easy to implement and is used in most recent superscalars.

One question you may be asking is how do we ever know which registers are the architectural registers if they are constantly changing? Most of the time when the program is executing, it does not matter. There are clearly cases, however, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register reside. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. Eventually all instructions in the pipeline will commit, and the mapping between the architecturally visible registers and physical registers will become stable. At that point, a subset of the physical registers contains the architecturally visible registers, and the value of any physical register not associated with an architectural register is unneeded. It is then easy to move the architectural registers to a fixed subset of physical registers so that the values can be communicated to another process.

Both register renaming and reorder buffers continue to be used in high-end processors, which now feature the ability to have as many as 100 or more instructions (including loads and stores waiting on the cache) in flight. Whether renaming or a reorder buffer is used, the key complexity bottleneck for a dynamically scheduled superscalar remains issuing bundles of instructions with dependences within the bundle. In particular, dependent instructions in an issue bundle must be issued with the assigned virtual registers of the instructions on which they depend. A strategy for instruction issue with register renaming similar to that used for multiple issue with reorder buffers (see page 205) can be deployed, as follows:

1. The issue logic reserves enough physical registers for the entire issue bundle (say, four registers for a four-instruction bundle with at most one register result per instruction).
2. The issue logic determines what dependences exist within the bundle. If a dependence does not exist within the bundle, the register renaming structure is used to determine the physical register that holds, or will hold, the result on which instruction depends. When no dependence exists within the bundle, the result is from an earlier issue bundle, and the register renaming table will have the correct register number.
3. If an instruction depends on an instruction that is earlier in the bundle, then the pre-reserved physical register in which the result will be placed is used to update the information for the issuing instruction.

Note that just as in the reorder buffer case, the issue logic must both determine dependences within the bundle and update the renaming tables in a single clock, and as before, the complexity of doing this for a larger number of instructions per clock becomes a chief limitation in the issue width.

The Challenge of More Issues per Clock

Without speculation, there is little motivation to try to increase the issue rate beyond two, three, or possibly four issues per clock because resolving branches would limit the average issue rate to a smaller number. Once a processor includes accurate branch prediction and speculation, we might conclude that increasing the issue rate would be attractive. Duplicating the functional units is straightforward assuming silicon capacity and power; the real complications arise in the issue step and correspondingly in the commit step. The Commit step is the dual of the issue step, and the requirements are similar, so let's take a look at what has to happen for a six-issue processor using register renaming.

Figure 3.29 shows a six-instruction code sequence and what the issue step must do. Remember that this must all occur in a single clock cycle, if the processor is to maintain a peak rate of six issues per clock! All the dependences must be detected,

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	`add x1,x2,x3`	p32	`add p32,p2,p3`	**x1-> p32**
2	`sub x1,x1,x2`	p33	`sub p33,p32,p2`	**x1->p33**
3	`add x2,x1,x2`	p34	`add p34,p33,x2`	**x2->p34**
4	`sub x1,x3,x2`	p35	`sub p35,p3,p34`	**x1->p35**
5	`add x1,x1,x2`	p36	`add p36,p35,p34`	**x1->p36**
6	`sub x1,x3,x1`	p37	`sub p37,p3,p36`	**x1->p37**

Figure 3.29 An example of six instructions to be issued in the same clock cycle and what has to happen. The instructions are shown in program order: 1–6; they are, however, issued in 1 clock cycle! The notation `pi` is used to refer to a physical register; the contents of that register at any point is determined by the renaming map. For simplicity, we assume that the physical registers holding the architectural registers `x1`, `x2`, and x3 are initially `p1`, `p2`, and `p3` (they could be any physical register). The instructions are issued with physical register numbers, as shown in column four. The rename map, which appears in the last column, shows how the map would change if the instructions were issued sequentially. The difficulty is that all this renaming and replacement of architectural registers by physical renaming registers happens effectively in 1 cycle, not sequentially. The issue logic must find all the dependences and "rewrite" the instruction in parallel.

the physical registers must be assigned, and the instructions must be rewritten using the physical register numbers: in one clock. This example makes it clear why issue rates have grown from 3–4 to only 4–8 in the past 20 years. The complexity of the analysis required during the issue cycle grows as the square of the issue width, and a new processor is typically targeted to have a higher clock rate than in the last generation! Because register renaming and the reorder buffer approaches are duals, the same complexities arise independent of the implementation scheme.

How Much to Speculate

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage. Speculation is not free. It takes time and energy, and the recovery of incorrect speculation further reduces performance. In addition, to support the higher instruction execution rate needed to benefit from speculation, the processor must have additional resources, which take silicon area and power. Finally, if speculation causes an exceptional event to occur, such as a cache or translation lookaside buffer (TLB) miss, the potential for significant performance loss increases, if that event would not have occurred without speculation.

To maintain most of the advantage while minimizing the disadvantages, most pipelines with speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a TLB miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less-than-excellent branch prediction.

In the 1990s the potential downsides of speculation were less obvious. As processors have evolved, the real costs of speculation have become more apparent, and the limitations of wider issue and speculation have been obvious. We return to this issue shortly.

Speculating Through Multiple Branches

In the examples we have considered in this chapter, it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculating on multiple branches simultaneously: (1) a very high branch frequency, (2) significant clustering of branches, and (3) long delays in functional units. In the first two cases, achieving high performance may mean that multiple branches are speculated, and it may even mean handling more than one branch per clock. Database programs and other less structured integer computations, often exhibit these properties, making speculation on multiple branches important. Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays.

Speculating on multiple branches slightly complicates the process of speculation recovery but is straightforward otherwise. As of 2017, no processor has yet combined full speculation with resolving multiple branches per cycle, and it is unlikely that the costs of doing so would be justified in terms of performance versus complexity and power.

Speculation and the Challenge of Energy Efficiency

What is the impact of speculation on energy efficiency? At first glance, one might argue that using speculation always decreases energy efficiency because whenever speculation is wrong, it consumes excess energy in two ways:

1. Instructions that are speculated and whose results are not needed generate excess work for the processor, wasting energy.
2. Undoing the speculation and restoring the state of the processor to continue execution at the appropriate address consumes additional energy that would not be needed without speculation.

Certainly, speculation will raise the power consumption, and if we could control speculation, it would be possible to measure the cost (or at least the dynamic power cost). But, if speculation lowers the execution time by more than it increases the average power consumption, then the total energy consumed may be less.

Thus, to understand the impact of speculation on energy efficiency, we need to look at how often speculation is leading to unnecessary work. If a significant number of unneeded instructions is executed, it is unlikely that speculation will

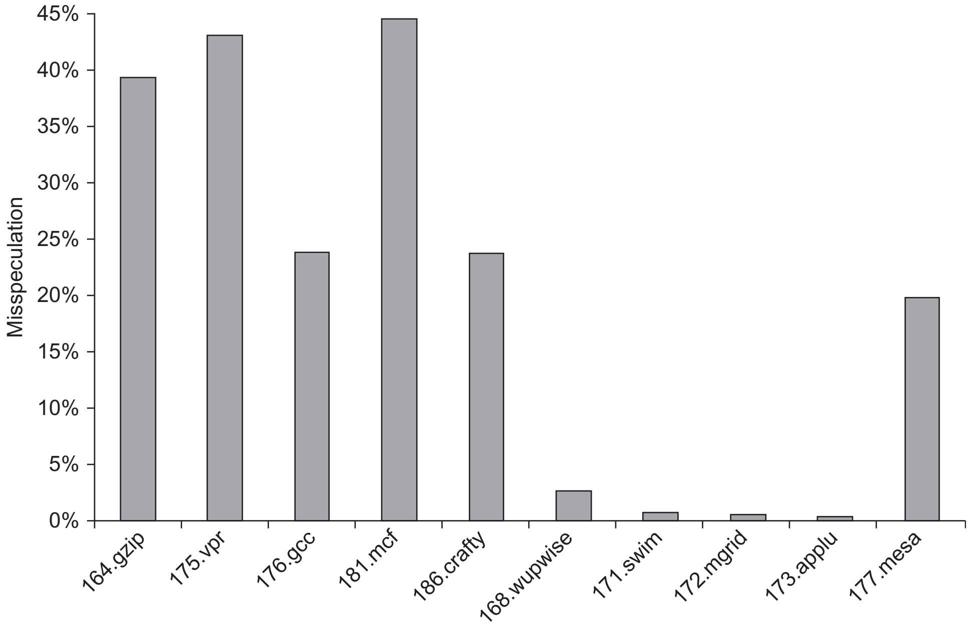


Figure 3.30 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

improve running time by a comparable amount. Figure 3.30 shows the fraction of instructions that are executed from misspeculation for a subset of the SPEC2000 benchmarks using a sophisticated branch predictor. As we can see, this fraction of executed misspeculated instructions is small in scientific code and significant (about 30% on average) in integer code. Thus it is unlikely that speculation is energy-efficient for integer applications, and the end of Dennard scaling makes imperfect speculation more problematic. Designers could avoid speculation, try to reduce the misspeculation, or think about new approaches, such as only speculating on branches that are known to be highly predictable.

Address Aliasing Prediction

Address aliasing prediction is a technique that predicts whether two stores or a load and a store refer to the same memory address. If two such references do not refer to the same address, then they may be safely interchanged. Otherwise, we must wait until the memory addresses accessed by the instructions are known. Because we need not actually predict the address values, only whether such values conflict, the prediction can be reasonably accurate with small predictors. Address prediction relies on the ability of a speculative processor to recover after a misprediction; that is, if the actual addresses that were predicted to be different (and thus not alias) turn out to be the same (and thus are aliases), the processor simply restarts the sequence, just as though it had mispredicted a branch. Address value speculation has been used in several processors already and may become universal in the future.

Address prediction is a simple and restricted form of *value prediction,* which attempts to predict the value that will be produced by an instruction. Value prediction could, if it were highly accurate, eliminate data flow restrictions and achieve higher rates of ILP. Despite many researchers focusing on value prediction in the past 15 years in dozens of papers, the results have never been sufficiently attractive to justify general value prediction in real processors.

3.10 Cross-Cutting Issues

Hardware Versus Software Speculation

The hardware-intensive approaches to speculation in this chapter and the software approaches of Appendix H provide alternative approaches to exploiting ILP. Some of the trade-offs, and the limitations, for these approaches are listed here:

- To speculate extensively, we must be able to disambiguate memory references. This capability is difficult to do at compile time for integer programs that contain pointers. In a hardware-based scheme, dynamic runtime disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.
- Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. These properties hold for many integer programs, where the misprediction rates for dynamic predictors are usually less than one-half of those for static predictors. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.
- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.
- Hardware-based speculation does not require compensation or bookkeeping code, which is needed by ambitious software speculation mechanisms.
- Compiler-based approaches may benefit from the ability to see further into the code sequence, resulting in better code scheduling than a purely hardware-driven approach.
- Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations

of an architecture. Although this advantage is the hardest to quantify, it may be the most important one in the long run. Interestingly, this was one of the motivations for the IBM 360/91. On the other hand, more recent explicitly parallel architectures, such as IA-64, have added flexibility that reduces the hardware dependence inherent in a code sequence.

The major disadvantage of supporting speculation in hardware is the complexity and additional hardware resources required. This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler.

Some designers have tried to combine the dynamic and compiler-based approaches to achieve the best of each. Such a combination can generate interesting and obscure interactions. For example, if conditional moves are combined with register renaming, a subtle side effect appears. A conditional move that is annulled must still copy a value to the destination register because it was renamed earlier in the instruction pipeline. These subtle interactions complicate the design and verification process and can also reduce performance.

The Intel Itanium processor was the most ambitious computer ever designed based on the software support for ILP and speculation. It did not deliver on the hopes of the designers, especially for general-purpose, nonscientific code. As designers' ambitions for exploiting ILP were reduced in light of the difficulties described on page 244, most architectures settled on hardware-based mechanisms with issue rates of three to four instructions per clock.

Speculative Execution and the Memory System

Inherent in processors that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if protection exceptions were taken, but also the benefits of speculative execution would be swamped by false exception overhead. Therefore the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss because, again, unnecessary stalls could overwhelm the benefits of speculation. Thus these processors must be matched with nonblocking caches.

In reality, the penalty of a miss that goes to DRAM is so large that speculated misses are handled only when the next level is on-chip cache (L2 or L3). Figure 2.5 on page 84 shows that for some well-behaved scientific programs, the compiler can sustain multiple outstanding L2 misses to cut the L2 miss penalty effectively. Once again, for this to work, the memory system behind the cache must match the goals of the compiler in number of simultaneous memory accesses.

3.11 Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

The topic we cover in this section, multithreading, is truly a cross-cutting topic, because it has relevance to pipelining and superscalars, to graphics processing units (Chapter 4), and to multiprocessors (Chapter 5). A *thread* is like a process in that it has state and a current program counter, but threads typically share the address space of a single process, allowing a thread to easily access data of other threads within the same process. Multithreading is a technique whereby multiple threads share a processor without requiring an intervening process switch. The ability to switch between threads rapidly is what enables multithreading to be used to hide pipeline and memory latencies.

In the next chapter, we will see how multithreading provides the same advantages in GPUs. Finally, Chapter 5 will explore the combination of multithreading and multiprocessing. These topics are closely interwoven because multithreading is a primary technique for exposing more parallelism to the hardware. In a strict sense, multithreading uses thread-level parallelism, and thus is properly the subject of Chapter 5, but its role in both improving pipeline utilization and in GPUs motivates us to introduce the concept here.

Although increasing performance by using ILP has the great advantage that it is reasonably transparent to the programmer, as we have seen, ILP can be quite limited or difficult to exploit in some applications. In particular, with reasonable instruction issue rates, cache misses that go to memory or off-chip caches are unlikely to be hidden by available ILP. Of course, when the processor is stalled waiting on a cache miss, the utilization of the functional units drops dramatically.

Because attempts to cover long memory stalls with more ILP have limited effectiveness, it is natural to ask whether other forms of parallelism in an application could be used to hide memory delays. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. Of course, many scientific applications contain natural parallelism because they often model the three-dimensional, parallel structure of nature, and that structure can be exploited by using separate threads. Even desktop applications that use modern Windows-based operating systems often have multiple active applications running, providing a source of parallelism.

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In contrast, a more general method to exploit *thread-level parallelism* (TLP) is with a multiprocessor that has multiple independent threads operating at once and in parallel. Multithreading, however, does not duplicate the entire processor as a multiprocessor does. Instead, multithreading shares most of the processor core among a set of threads, duplicating only private state, such as the registers and program counter. As we will see in Chapter 5, many recent processors incorporate both multiple processor cores on a single chip and provide multithreading within each core.

Duplicating the per-thread state of a processor core means creating a separate register file and a separate PC for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles. Of course, for multithreading hardware to achieve performance improvements, a program must contain multiple threads (we sometimes say that the application is multithreaded) that could execute in concurrent fashion. These threads are identified either by a compiler (typically from a language with parallelism constructs) or by the programmer.

There are three main hardware approaches to multithreading: fine-grained, coarse-grained, and simultaneous. *Fine-grained multithreading* switches between threads on each clock cycle, causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls because instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles. The primary disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread because a thread that is ready to execute without stalls will be delayed by instructions from other threads. It trades an increase in multithreaded throughput for a loss in the performance (as measured by latency) of a single thread.

The SPARC T1 through T5 processors (originally made by Sun, now made by Oracle and Fujitsu) use fine-grained multithreading. These processors were targeted at multithreaded workloads such as transaction processing and web services. The T1 supported 8 cores per processor and 4 threads per core, while the T5 supports 16 cores and 128 threads per core. Later versions (T2–T5) also supported 4–8 processors. The NVIDIA GPUs, which we look at in the next chapter, also make use of fine-grained multithreading.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two or three cache misses. Because instructions from other threads will be issued only when a thread encounters a costly stall, coarse-grained multithreading relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread.

Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline will see a bubble before the new thread begins executing. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time. Several research projects have explored coarse-grained multithreading, but no major current processors use this technique.

The most common implementation of multithreading is called *simultaneous multithreading* (SMT). Simultaneous multithreading is a variation on fine-grained multithreading that arises naturally when fine-grained multithreading is implemented on top of a multiple-issue, dynamically scheduled processor. As with other forms of multithreading, SMT uses thread-level parallelism to hide long-latency events in a processor, thereby increasing the usage of the functional units. The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 3.31 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

- A superscalar with no multithreading support
- A superscalar with coarse-grained multithreading

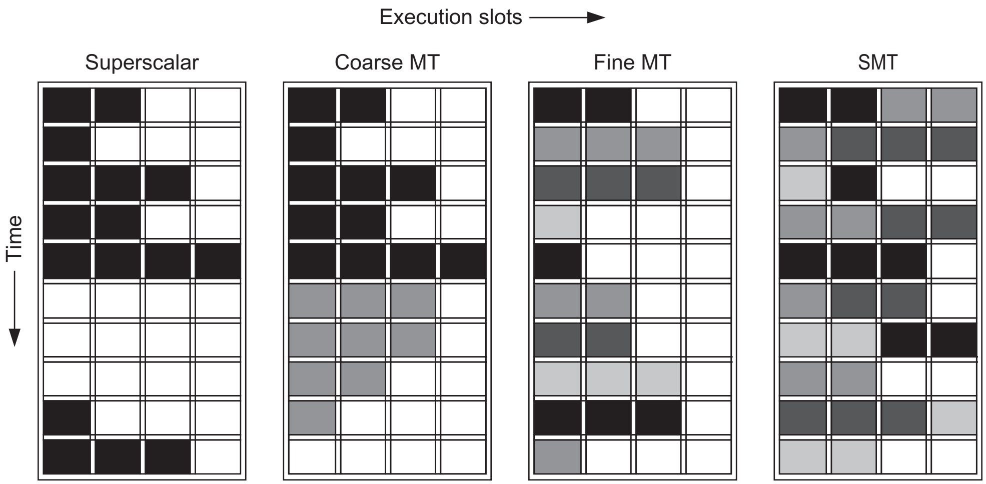


Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP, including ILP to hide memory latency. Because of the length of L2 and L3 cache misses, much of the processor can be left idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. This switching reduces the number of completely idle clock cycles. In a coarse-grained multithreaded processor, however, thread switching occurs only when there is a stall. Because the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads can eliminate fully empty slots. In addition, because the issuing thread is changed on every clock cycle, longer latency operations can be hidden. Because instruction issue and execution are connected, a thread can issue only as many instructions as are ready. With a narrow issue width, this is not a problem (a cycle is either occupied or not), which is why fine-grained multithreading works perfectly for a single issue processor, and SMT would make no sense. Indeed, in the Sun T2, there are two issues per clock, but they are from different threads. This eliminates the need to implement the complex dynamic scheduling approach and relies instead on hiding latency with more threads.

If one implements fine-grained threading on top of a multiple-issue, dynamically schedule processor, the result is SMT. In all existing SMT implementations, all issues come from one thread, although instructions from different threads can initiate execution in the same cycle, using the dynamic scheduling hardware to determine what instructions are ready. Although Figure 3.31 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in wider issue, dynamically scheduled processors.

Simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the mechanism, including a large virtual register set. Multithreading can be built on top of an out-of-order processor by adding a per-thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

Effectiveness of Simultaneous Multithreading on Superscalar Processors

A key question is, how much performance can be gained by implementing SMT? When this question was explored in 2000–2001, researchers assumed that dynamic superscalars would get much wider in the next five years, supporting six to eight issues per clock with speculative dynamic scheduling, many simultaneous loads and stores, large primary caches, and four to eight contexts with simultaneous issue and retirement from multiple contexts. No processor has gotten close to this combination.

As a result, simulation research results that showed gains for multiprogrammed workloads of two or more times are unrealistic. In practice, the existing implementations of SMT offer only two to four contexts with fetching and issue from only one, and up to four issues per clock. The result is that the gain from SMT is also more modest.

Esmaeilzadeh et al. (2011) did an extensive and insightful set of measurements that examined both the performance and energy benefits of using SMT in a single i7 920 core running a set of multithreaded applications. The Intel i7 920 supported SMT with two threads per core, as does the recent i7 6700. The changes between the i7 920 and the 6700 are relatively small and are unlikely to significantly change the results as shown in this section.

The benchmarks used consist of a collection of parallel scientific applications and a set of multithreaded Java programs from the DaCapo and SPEC Java suite, as summarized in Figure 3.32. Figure 3.31 shows the ratios of performance and energy efficiency for these benchmarks when run on one core of a i7 920 with SMT turned off and on. (We plot the energy efficiency ratio, which is the inverse of energy consumption, so that, like speedup, a higher ratio is better.)

The harmonic mean of the speedup for the Java benchmarks is 1.28, despite the two benchmarks that see small gains. These two benchmarks, pjbb2005 and tradebeans, while multithreaded, have limited parallelism. They are included because they are typical of a multithreaded benchmark that might be run on an SMT processor with the hope of extracting some performance, which they find in limited amounts. The PARSEC benchmarks obtain somewhat better speedups than the full set of Java benchmarks (harmonic mean of 1.31). If tradebeans and pjbb2005 were omitted, the Java workload would actually have significantly better speedup (1.39) than the PARSEC benchmarks. (See the discussion of the implication of using harmonic mean to summarize the results in the caption of Figure 3.33.)

Energy consumption is determined by the combination of speedup and increase in power consumption. For the Java benchmarks, on average, SMT delivers the same energy efficiency as non-SMT (average of 1.0), but it is brought down by the two poor performing benchmarks; without pjbb2005 and tradebeans, the average energy efficiency for the Java benchmarks is 1.06, which is almost as good as the PARSEC benchmarks. In the PARSEC benchmarks, SMT reduces energy by $1-(1/1.08)=7\%$. Such energy-reducing performance enhancements are *very difficult* to find. Of course, the static power associated with SMT is paid in both cases, thus the results probably slightly overstate the energy gains.

These results clearly show that SMT with extensive support in an aggressive speculative processor can improve performance in an energy-efficient fashion. In 2011, the balance between offering multiple simpler cores and fewer more sophisticated cores has shifted in favor of more cores, with each core typically being a three- to four-issue superscalar with SMT supporting two to four threads. Indeed, Esmaeilzadeh et al. (2011) show that the energy improvements from SMT are even larger on the Intel i5 (a processor similar to the i7, but with smaller caches and a

blackscholes	Prices a portfolio of options with the Black-Scholes PDE
bodytrack	Tracks a markerless human body
canneal	Minimizes routing cost of a chip with cache-aware simulated annealing
facesim	Simulates motions of a human face for visualization purposes
ferret	Search engine that finds a set of images similar to a query image
fluidanimate	Simulates physics of fluid motion for animation with SPH algorithm
raytrace	Uses physical simulation for visualization
streamcluster	Computes an approximation for the optimal clustering of data points
swaptions	Prices a portfolio of swap options with the Heath–Jarrow–Morton framework
vips	Applies a series of transformations to an image
x264	MPG-4 AVC/H.264 video encoder
eclipse	Integrated development environment
lusearch	Text search tool
sunflow	Photo-realistic rendering system
tomcat	Tomcat servlet container
tradebeans	Tradebeans Daytrader benchmark
xalan	An XSLT processor for transforming XML documents
pjbb2005	Version of SPEC JBB2005 (but fixed in problem size rather than time)

Figure 3.32 The parallel benchmarks used here to examine multithreading, as well as in Chapter 5 to examine multiprocessing with an i7. The top half of the chart consists of PARSEC benchmarks collected by Bienia et al. (2008). The PARSEC benchmarks are meant to be indicative of compute-intensive, parallel applications that would be appropriate for multicore processors. The lower half consists of multithreaded Java benchmarks from the DaCapo collection (see Blackburn et al., 2006) and pjbb2005 from SPEC. All of these benchmarks contain some parallelism; other Java benchmarks in the DaCapo and SPEC Java workloads use multiple threads but have little or no true parallelism and, hence, are not used here. See Esmaeilzadeh et al. (2011) for additional information on the characteristics of these benchmarks, relative to the measurements here and in Chapter 5.

lower clock rate) and the Intel Atom (an 80x86 processor originally designed for the netbook and PMD market, now focused on low-end PCs, and described in Section 3.13).

3.12 Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53

In this section, we explore the design of two multiple issue processors: the ARM Cortex-A53 core, which is used as the basis for several tablets and cell phones, and the Intel Core i7 6700, a high-end, dynamically scheduled, speculative processor intended for high-end desktops and server applications. We begin with the simpler processor.

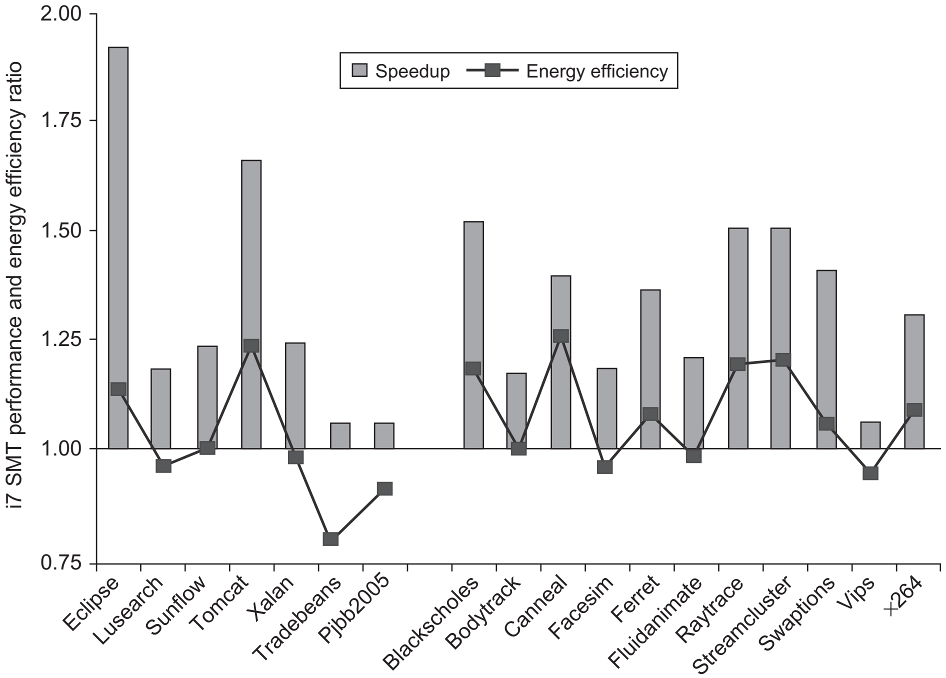


Figure 3.33 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was the same). The energy efficiency averages 0.99 and 1.07, respectively (using the harmonic mean). Recall that anything above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power. Two of the Java benchmarks experience little speedup and have significant negative energy efficiency because of this issue. Turbo Boost is off in all cases. These data were collected and analyzed by Esmaeilzadeh et al. (2011) using the Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine and the gcc v4.4.1 native compiler.

The ARM Cortex-A53

The A53 is a dual-issue, statically scheduled superscalar with dynamic issue detection, which allows the processor to issue two instructions per clock. Figure 3.34 shows the basic pipeline structure of the pipeline. For nonbranch, integer instructions, there are eight stages: F1, F2, D1, D2, D3/ISS, EX1, EX2, and WB, as described in the caption. The pipeline is in order, so an instruction can initiate execution only when its results are available and when proceeding instructions have initiated. Thus, if the next two instructions are dependent, both can proceed to the appropriate execution pipeline, but they will be serialized when they get to the beginning of that pipeline. When the scoreboard-based issue logic indicates that the result from the first instruction is available, the second instruction can issue.

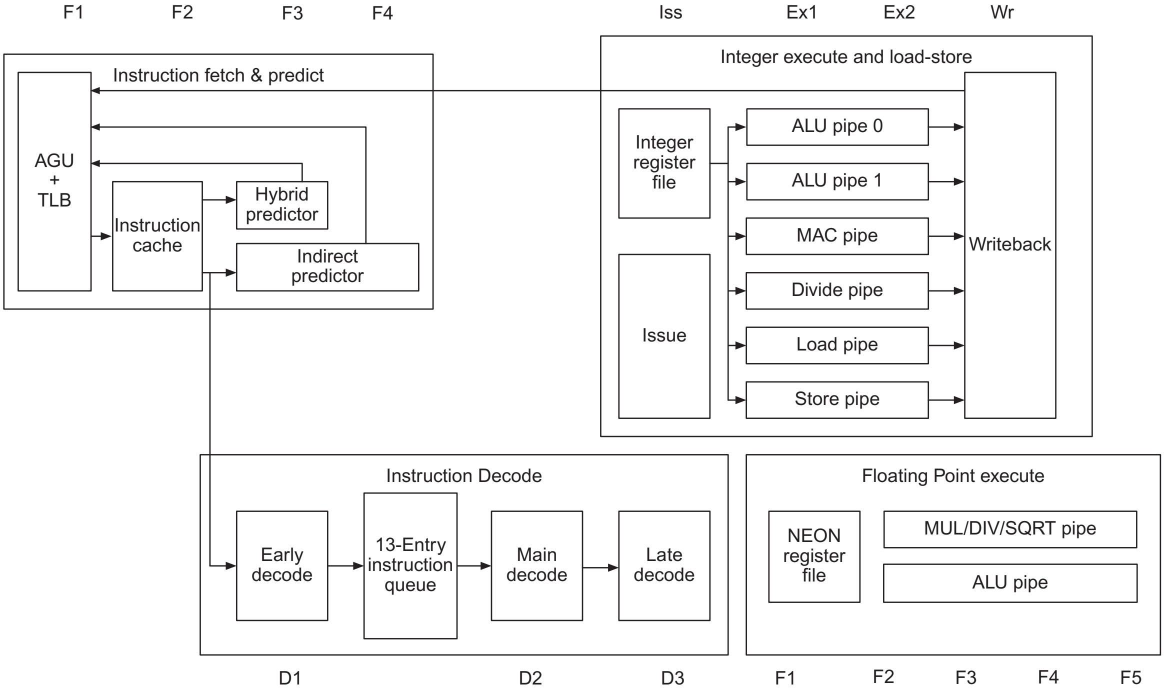

F1
F2
F3
F4
Iss
Ex1
Ex2
Wr
Instruction fetch & predict
AGU
+
TLB
Instruction cache
Hybrid predictor
Indirect predictor
Integer execute and load-store
Integer register file
Issue
ALU pipe 0
ALU pipe 1
MAC pipe
Divide pipe
Load pipe
Store pipe
Writeback
Instruction Decode
Early decode
13-Entry instruction queue
Main decode
Late decode
Floating Point execute
NEON register file
MUL/DIV/SQRT pipe
ALU pipe
D1
D2
D3
F1
F2
F3
F4
F5

Figure 3.34 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

The four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of four predictors:

1. A single-entry branch target cache containing two instruction cache fetches (the next two instructions following the branch, assuming the prediction is correct). This target cache is checked during the first fetch cycle, if it hits; then the next two instructions are supplied from the target cache. In case of a hit and a correct prediction, the branch is executed with no delay cycles.
2. A 3072-entry hybrid predictor, used for all instructions that do not hit in the branch target cache, and operating during F3. Branches handled by this predictor incur a 2-cycle delay.
3. A 256-entry indirect branch predictor that operates during F4; branches predicted by this predictor incur a three-cycle delay when predicted correctly.
4. An 8-deep return stack, operating during F4 and incurring a three-cycle delay.

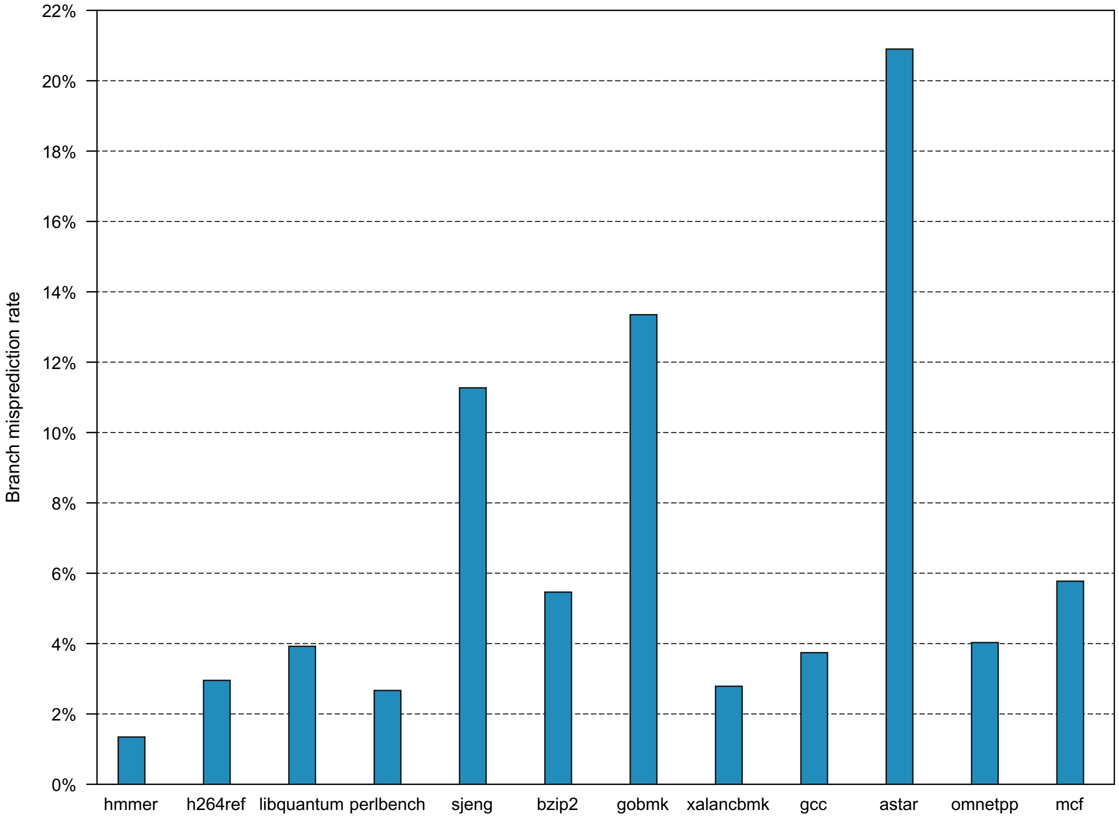


Figure 3.35 Misprediction rate of the A53 branch predictor for SPECint2006.

Branch decisions are made in ALU pipe 0, resulting in a branch misprediction penalty of 8 cycles. Figure 3.35 shows the misprediction rate for SPECint2006. The amount of work that is wasted depends on both the misprediction rate and the issue rate sustained during the time that the mispredicted branch was followed. As Figure 3.36 shows, wasted work generally follows the misprediction rate, though it may be larger or occasionally shorter.

Performance of the A53 Pipeline

The A53 has an ideal CPI of 0.5 because of its dual-issue structure. Pipeline stalls can arise from three sources:

1. Functional hazards, which occur because two adjacent instructions selected for issue simultaneously use the same functional pipeline. Because the A53 is statically scheduled, the compiler should try to avoid such conflicts. When such

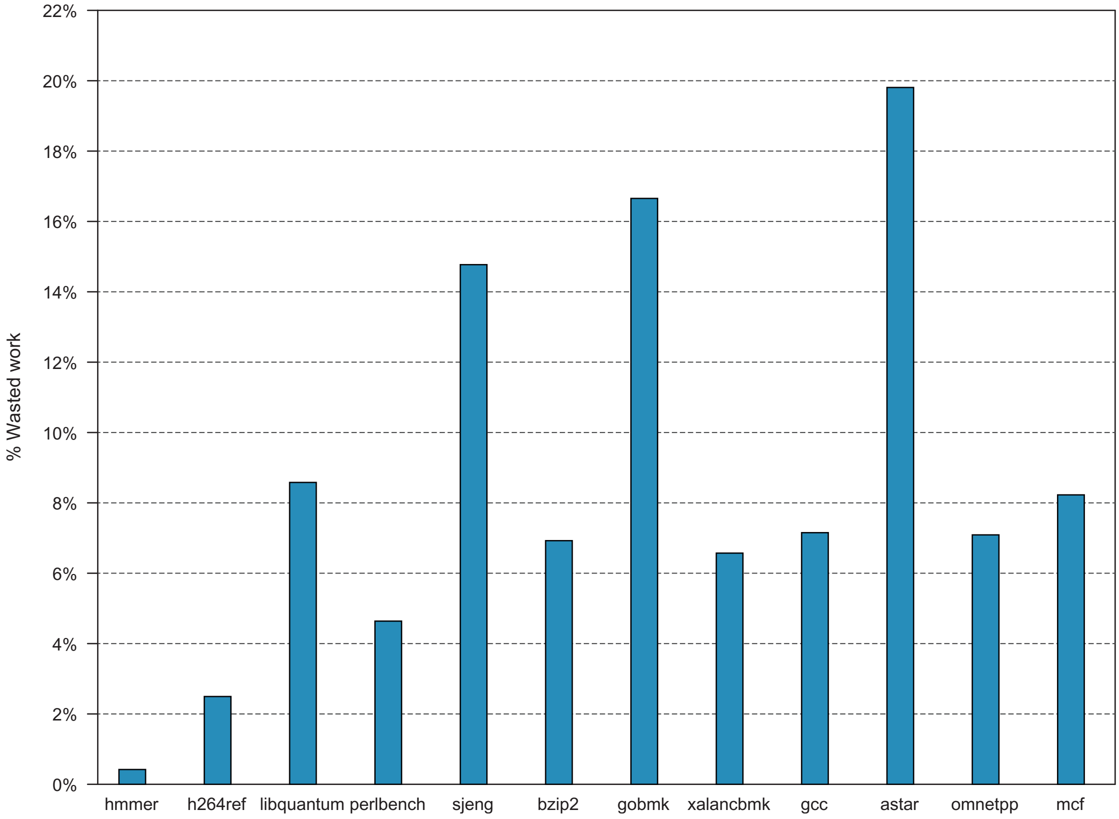


Figure 3.36 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

instructions appear sequentially, they will be serialized at the beginning of the execution pipeline, when only the first instruction will begin execution.

2. Data hazards, which are detected early in the pipeline and may stall either both instructions (if the first cannot issue, the second is always stalled) or the second of a pair. Again, the compiler should try to prevent such stalls when possible.
3. Control hazards, which arise only when branches are mispredicted.

Both TLB misses and cache misses also cause stalls. On the instruction side, a TLB or cache miss causes a delay in filling the instruction queue, likely leading to a downstream stall of the pipeline. Of course, this depends on whether it is an L1 miss, which might be largely hidden if the instruction queue was full at the time of the miss, or an L2 miss, which takes considerably longer. On the data side, a cache or TLB miss will cause the pipeline to stall because the load or store that

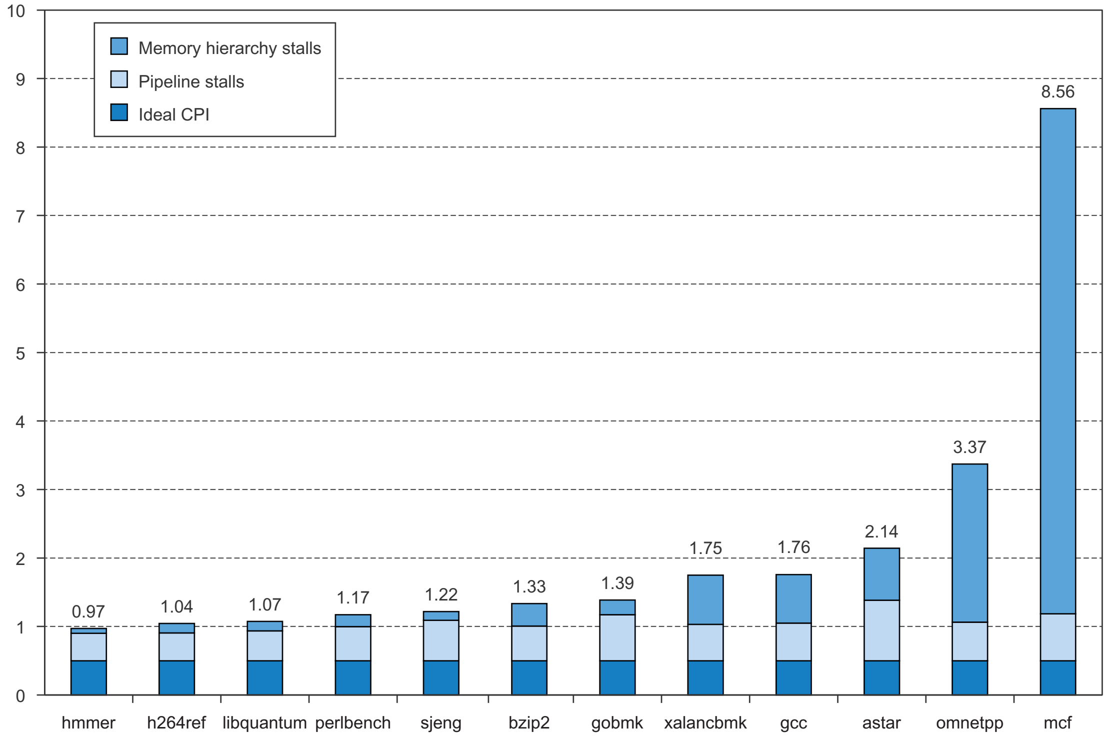


Figure 3.37 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

caused the miss cannot proceed down the pipeline. All other subsequent instructions will thus be stalled. Figure 3.37 shows the CPI and the estimated contributions from various sources.

The A53 uses a shallow pipeline and a reasonably aggressive branch predictor, leading to modest pipeline losses, while allowing the processor to achieve high clock rates at modest power consumption. In comparison with the i7, the A53 consumes approximately 1/200 the power for a quad core processor!

The Intel Core i7

The i7 uses an aggressive out-of-order speculative microarchitecture with deep pipelines with the goal of achieving high instruction throughput by combining multiple issue and high clock rates. The first i7 processor was introduced in 2008; the i7 6700 is the sixth generation. The basic structure of the i7 is similar, but successive generations have enhanced performance by changing cache strategies (e.g., the aggressiveness of prefetching), increasing memory bandwidth, expanding the number of instructions in flight, enhancing branch prediction, and improving graphics support. The early i7 microarchitectures used reservations stations and reorder buffers for their out-of-order, speculative pipeline. Later microarchitectures, including the i7 6700, use register renaming, with the reservations stations acting as functional unit queues and the reorder buffer simply tracking control information.

Figure 3.38 shows the overall structure of the i7 pipeline. We will examine the pipeline by starting with instruction fetch and continuing on to instruction commit, following steps labeled in the figure.

1. Instruction fetch—The processor uses a sophisticated multilevel branch predictor to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 17 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer—In this step, a process called macro-op fusion is executed. *Macro-op fusion* takes instruction combinations such as compare followed by a branch and fuses them into a single operation, which can issue and dispatch as one instruction. Only certain special cases can be fused, since we must know that the only use of the first result is by the second instruction (i.e., compare and branch). In a study of the Intel Core architecture (which has many fewer buffers), Bird et al. (2007) discovered that macrofusion had a significant impact on the performance of integer programs resulting in an 8%–10% average increase in performance with a few programs showing negative results. There was little impact on FP programs; in fact, about half of the SPECFP benchmarks showed negative results from macro-op fusion. The predecode stage also breaks the 16 bytes into individual x86 instructions. This predecode is nontrivial because the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions (including some fused instructions) are placed into the instruction queue.
3. Micro-op decode—Individual x86 instructions are translated into micro-ops. Micro-ops are simple RISC-V-like instructions that can be executed directly by the pipeline; this approach of translating the x86 instruction set into simple operations that are more easily pipelined was introduced in the Pentium Pro in 1997 and has been used since. Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 64-entry micro-op buffer.

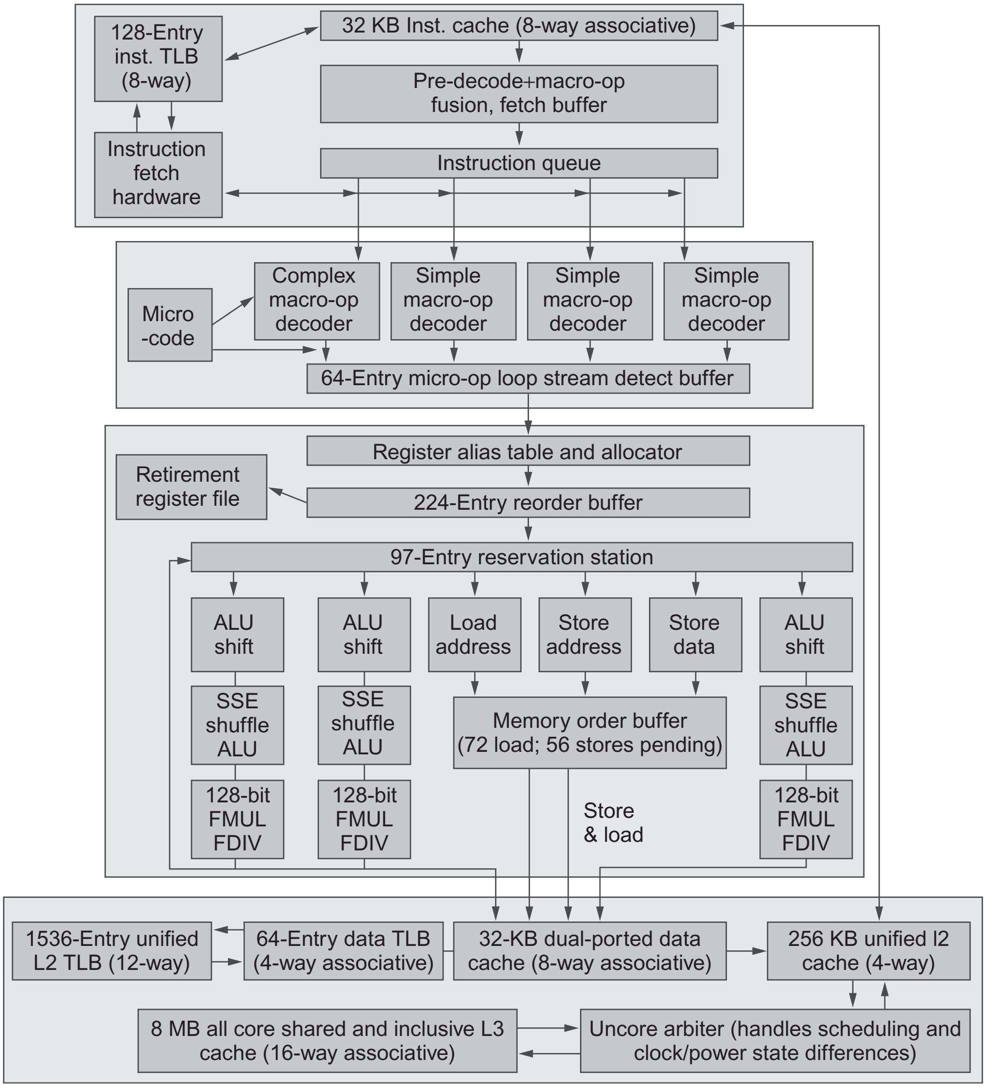

128-Entry inst. TLB (8-way)
32 KB Inst. cache (8-way associative)
Pre-decode+macro-op fusion, fetch buffer
Instruction fetch hardware
Instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
64-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
224-Entry reorder buffer
97-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer (72 load; 56 stores pending)
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
1536-Entry unified L2 TLB (12-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (4-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Figure 3.38 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles likely due to the time to reset the branch predictor. The six independent functional units can each begin execution of a ready micro-op in the same cycle. Up to four micro-ops can be processed in the register renaming table.

4. The micro-op buffer preforms *loop stream detection* and *microfusion*—If there is a small sequence of instructions (less than 64 instructions) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated. Microfusion combines

instruction pairs such as ALU operation and a dependent store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer. Micro-op fusion produces smaller gains for integer programs and larger ones for FP, but the results vary widely. The different results for integer and FP programs with macro and micro fusion, probably arise from the patterns recognized and fused and the frequency of occurrence in integer versus FP programs. In the i7, which has a much larger number of reorder buffer entries, the benefits from both techniques are likely to be smaller.

5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations. Up to four micro-ops can be processed every clock cycle; they are assigned the next available reorder buffer entries.
6. The i7 uses a centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. Micro-ops are executed by the individual function units, and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

In addition to the changes in the branch predictor, the major changes between the first generation i7 (the 920, Nehalem microarchitecture) and the sixth generation (i7 6700, Skylake microarchitecture) are in the sizes of the various buffers, renaming registers, and resources so as to allow many more outstanding instructions. Figure 3.39 summarizes these differences.

Performance of the i7

In earlier sections, we examined the performance of the i7's branch predictor and also the performance of SMT. In this section, we look at single-thread pipeline performance. Because of the presence of aggressive speculation as well as nonblocking caches, it is difficult to accurately attribute the gap between idealized performance and actual performance. The extensive queues and buffers on the 6700 reduce the probability of stalls because of a lack of reservation stations, renaming registers, or reorder buffers significantly. Indeed, even on the earlier i7 920 with notably fewer buffers, only about 3% of the loads were delayed because no reservation station was available.

Resource	**i7 920 (Nehalem)**	**i7 6700 (Skylake)**
Micro-op queue (per thread)	28	64
Reservation stations	36	97
Integer registers	NA	180
FP registers	NA	168
Outstanding load buffer	48	72
Outstanding store buffer	32	56
Reorder buffer	128	256

Figure 3.39 The buffers and queues in the first generation i7 and the latest generation i7. Nehalem used a reservation station plus reorder buffer organization. In later microarchitectures, the reservation stations serve as scheduling resources, and register renaming is used rather than the reorder buffer; the reorder buffer in the Skylake microarchitecture serves only to buffer control information. The choices of the size of various buffers and renaming registers, while appearing sometimes arbitrary, are likely based on extensive simulation.

Thus most losses come either from branch mispredicts or cache misses. The cost of a branch mispredict is 17 cycles, whereas the cost of an L1 miss is about 10 cycles. An L2 miss is slightly more than three times as costly as an L1 miss, and an L3 miss costs about 13 times what an L1 miss costs (130–135 cycles). Although the processor will attempt to find alternative instructions to execute during L2 and L3 misses, it is likely that some of the buffers will fill before a miss completes, causing the processor to stop issuing instructions.

Figure 3.40 shows the overall CPI for the 19 SPECCPUint2006 benchmarks compared to the CPI for the earlier i7 920. The average CPI on the i7 6700 is 0.71, whereas it is almost 1.5 times better on the i7 920, at 1.06. This difference derives from improved branch prediction and a reduction in the demand miss rates (see Figure 2.26 on page 135).

To understand how the 6700 achieves the significant improvement in CPI, let's look at the benchmarks that achieve the largest improvement. Figure 3.41 shows the five benchmarks that have a CPI ratio on the 920 that is at least 1.5 times higher than that of the 6700. Interestingly, three other benchmarks show a significant improvement in branch prediction accuracy (1.5 or more); however, those three benchmarks (HMMER, LIBQUANTUM, and SJENG) show equal or slightly higher L1 demand miss rates on the i7 6700. These misses likely arise because the aggressive prefetching is replacing cache blocks that are actually used. This type of behavior reminds designers of the challenges of maximizing performance in complex speculative multiple issue processors: rarely can significant performance be achieved by tuning only one part of the microarchitecture!

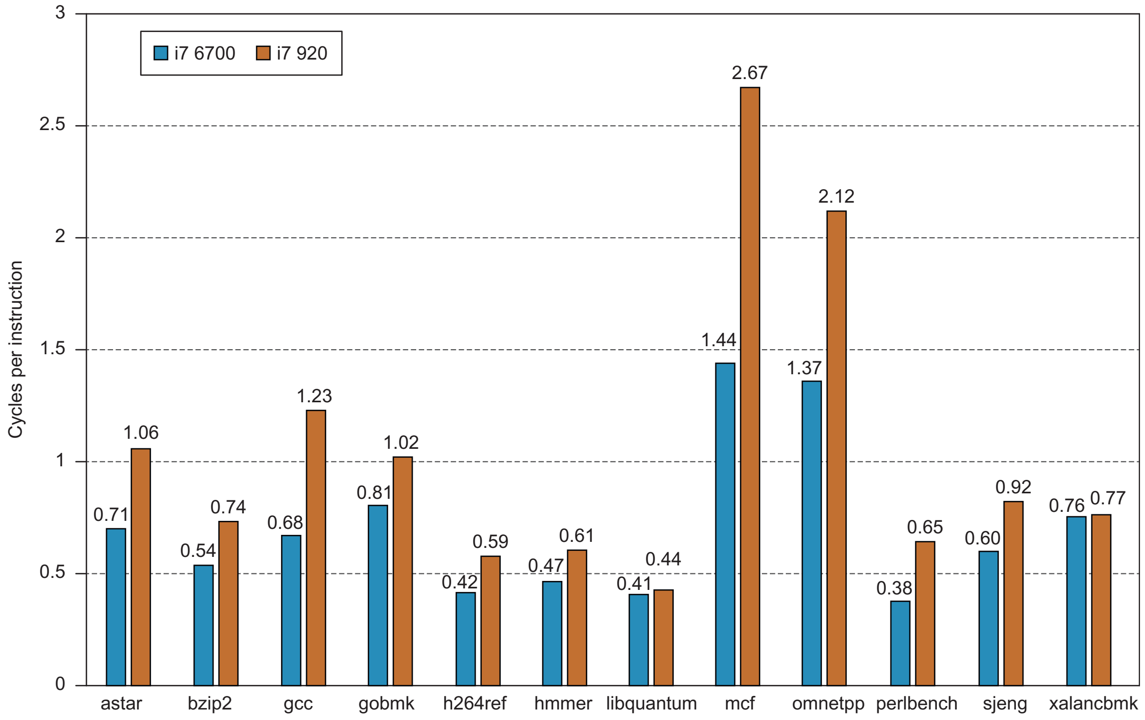


Figure 3.40 The CPI for the SPECCPUint2006 benchmarks on the i7 6700 and the i7 920. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

Benchmark	CPI ratio (920/6700)	Branch mispredict ratio (920/6700)	L1 demand miss ratio (920/6700)
ASTAR	1.51	1.53	2.14
GCC	1.82	2.54	1.82
MCF	1.85	1.27	1.71
OMNETPP	1.55	1.48	1.96
PERLBENCH	1.70	2.11	1.78

Figure 3.41 An analysis of the five integer benchmarks with the largest performance gap between the i7 6700 and 920. These five benchmarks show an improvement in the branch prediction rate and a reduction in the L1 demand miss rate.

3.13 Fallacies and Pitfalls

Our few fallacies focus on the difficulty of predicting performance and energy efficiency and extrapolating from single measures such as clock rate or CPI. We also show that different architectural approaches can have radically different behaviors for different benchmarks.

Fallacy *It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.*

Intel offers a processor for the low-end Netbook and PMD space called the Atom 230, which implements both the 64-bit and 32-bit versions of the x86 architecture. The Atom is a statically scheduled, 2-issue superscalar, quite similar in its microarchitecture to the ARM A8, a single-core predecessor of the A53. Interestingly, both the Atom 230 and the Core i7 920 have been fabricated in the same 45 nm Intel technology. Figure 3.42 summarizes the Intel Core i7 920, the ARM Cortex-A8, and the Intel Atom 230. These similarities provide a rare opportunity to directly compare two radically different microarchitectures for the same instruction set while holding constant the underlying fabrication technology. Before we do the comparison, we need to say a little more about the Atom 230.

The Atom processors implement the x86 architecture using the standard technique of translating x86 instructions into RISC-like instructions (as every x86 implementation since the mid-1990s has done). Atom uses a slightly more powerful microoperation, which allows an arithmetic operation to be paired with a load or a store; this capability was added to later i7s by the use of macrofusion. This means that on average for a typical instruction mix, only 4% of the instructions require more than one microoperation. The microoperations are then executed in a 16-deep pipeline capable of issuing two instructions per clock, in order, as in the ARM A8. There are dual-integer ALUs, separate pipelines for FP add and other FP operations, and two memory operation pipelines, supporting more general dual execution than the ARM A8 but still limited by the in-order issue capability. The Atom 230 has a 32 KiB instruction cache and a 24 KiB data cache, both backed by a shared 512 KiB L2 on the same die. (The Atom 230 also supports multithreading with two threads, but we will consider only single-threaded comparisons.)

We might expect that these two processors, implemented in the same technology and with the same instruction set, would exhibit predictable behavior, in terms of relative performance and energy consumption, meaning that power and performance would scale close to linearly. We examine this hypothesis using three sets of benchmarks. The first set is a group of Java single-threaded benchmarks that come from the DaCapo benchmarks and the SPEC JVM98 benchmarks (see Esmaeilzadeh et al. (2011) for a discussion of the benchmarks and measurements). The second and third sets of benchmarks are from SPEC CPU2006 and consist of the integer and FP benchmarks, respectively.

		Intel i7 920	**ARM A8**	**Intel Atom 230**
Area	**Specific characteristic**	**Four cores, each with FP**	**One core, no FP**	**One core, with FP**
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
Memory system	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KiB/32 KiB 256 KiB 2–8 MiB	Two-level 16/16 or 32/32 KiB 128 KiB–1 MiB	Two-level 32/24 KiB 512 KiB
	Peak memory BW	17 GB/s	12 GB/sec	8 GB/s
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipe line scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4 K global history 8-entry return stack	Two-level

Figure 3.42 An overview of the four-core Intel i7 920, an example of a typical ARM A8 processor chip (with a 256 MiB L2, 32 KiB L1s, and no floating point), and the Intel ARM 230, clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

As we can see in Figure 3.43, the i7 significantly outperforms the Atom. All benchmarks are at least four times faster on the i7, two SPECFP benchmarks are over 10 times faster, and one SPECINT benchmark runs over eight times faster! Because the ratio of clock rates of these two processors is 1.6, most of the advantage comes from a much lower CPI for the i7 920: a factor of 2.8 for the Java benchmarks, a factor of 3.1 for the SPECINT benchmarks, and a factor of 4.3 for the SPECFP benchmarks.

But the average power consumption for the i7 920 is just under 43 W, while the average power consumption of the Atom is 4.2 W, or about one-tenth of the power! Combining the performance and power leads to an energy efficiency advantage for the Atom that is typically more than 1.5 times better and often 2 times better! This comparison of two processors using the same underlying technology makes it clear that the performance advantages of an aggressive

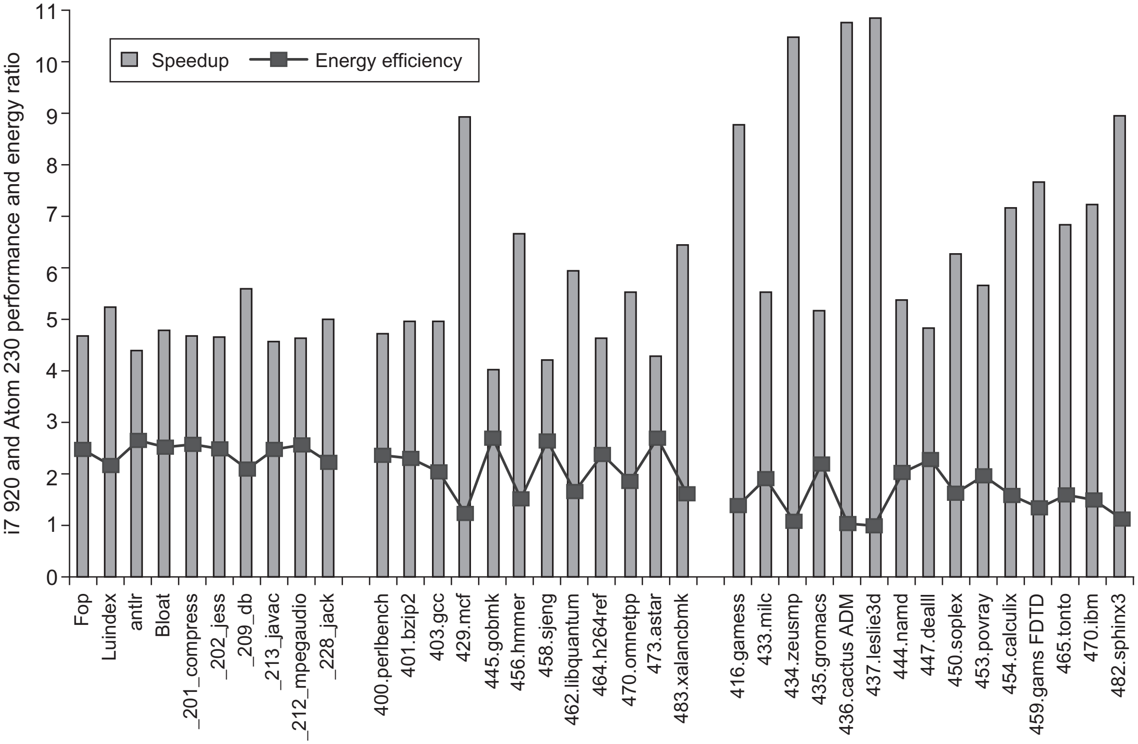


Figure 3.43 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power-efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. The data shown here were collected by Esmaeilzadeh et al. (2011). The SPEC benchmarks were compiled with optimization using the standard Intel compiler, while the Java benchmarks use the Sun (Oracle) Hotspot Java VM. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.

superscalar with dynamic scheduling and speculation come with a *significant disadvantage* in energy efficiency.

Fallacy *Processors with lower CPIs will always be faster.*

Fallacy *Processors with faster clock rates will always be faster.*

The key is that it is the product of CPI and clock rate that determines performance. A high clock rate obtained by deeply pipelining the processor must maintain a low CPI to get the full benefit of the faster clock. Similarly, a simple processor with a high clock rate but a low CPI may be slower.

As we saw in the previous fallacy, performance and energy efficiency can diverge significantly among processors designed for different environments even when they have the same ISA. In fact, large differences in performance can show up even within a family of processors from the same company all designed for high-end applications. Figure 3.44 shows the integer and FP performance of two different implementations of the x86 architecture from Intel, as well as a version of the Itanium architecture, also by Intel.

The Pentium 4 was the most aggressively pipelined processor ever built by Intel. It used a pipeline with over 20 stages, had seven functional units, and cached micro-ops rather than x86 instructions. Its relatively inferior performance, given the aggressive implementation, was a clear indication that the attempt to exploit more ILP (there could easily be 50 instructions in flight) had failed. The Pentium's power consumption was similar to the i7, although its transistor count was lower, as its primary caches were half as large as the i7, and it included only a 2 MiB secondary cache with no tertiary cache.

The Intel Itanium is a VLIW-style architecture, which despite the potential decrease in complexity compared to dynamically scheduled superscalars, never attained competitive clock rates with the mainline x86 processors (although it appears to achieve an overall CPI similar to that of the i7). In examining these results, the reader should be aware that they use different implementation technologies, giving the i7 an advantage in terms of transistor speed and hence clock rate for an equivalently pipelined processor. Nonetheless, the wide variation in

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

Figure 3.44 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks; the Power column is the thermal design power with estimates for only one core active in the multicore cases.

performance—more than three times between the Pentium and i7—is astonishing. The next pitfall explains where a significant amount of this advantage comes from.

Pitfall *Sometimes bigger and dumber is better.*

Much of the attention in the early 2000s went to building aggressive processors to exploit ILP, including the Pentium 4 architecture, which used the deepest pipeline ever seen in a microprocessor, and the Intel Itanium, which had the highest peak issue rate per clock ever seen. What quickly became clear was that the main limitation in exploiting ILP often turned out to be the memory system. Although speculative out-of-order pipelines were fairly good at hiding a significant fraction of the 10- to 15-cycle miss penalties for a first-level miss, they could do very little to hide the penalties for a second-level miss that, when going to main memory, were likely to be 50–100 clock cycles.

The result was that these designs never came close to achieving the peak instruction throughput despite the large transistor counts and extremely sophisticated and clever techniques. Section 3.15 discusses this dilemma and the turning away from more aggressive ILP schemes to multicore, but there was another change that exemplified this pitfall. Instead of trying to hide even more memory latency with ILP, designers simply used the transistors to build much larger caches. Both the Itanium 2 and the i7 use three-level caches compared to the two-level cache of the Pentium 4, and the third-level caches are 9 and 8 MiB compared to the 2 MiB second-level cache of the Pentium 4. Needless to say, building larger caches is a lot easier than designing the 20+-stage Pentium 4 pipeline, and based on the data in Figure 3.44, doing so seems to be more effective.

Pitfall *And sometimes smarter is better than bigger and dumber.*

One of the more surprising results of the past decade has been in branch prediction. The emergence of hybrid tagged predictors has shown that a more sophisticated predictor can outperform the simple gshare predictor with the same number of bits (see Figure 3.8 on page 171). One reason this result is so surprising is that the tagged predictor actually stores fewer predictions, because it also consumes bits to store tags, whereas gshare has only a large array of predictions. Nonetheless, it appears that the advantage gained by not misusing a prediction for one branch on another branch more than justifies the allocation of bits to tags versus predictions.

Pitfall *Believing that there are large amounts of ILP available, if only we had the right techniques.*

The attempts to exploit large amounts of ILP failed for several reasons, but one of the most important ones, which some designers did not initially accept, is that it is hard to find large amounts of ILP in conventionally structured programs, even with speculation. A famous study by David Wall in 1993 (see Wall, 1993) analyzed the amount of ILP available under a variety of idealistic conditions. We summarize his results for a processor configuration with roughly five to ten times the capability of the most advanced processors in 2017. Wall's study extensively documented a variety of different approaches, and the reader interested in the challenge of exploiting ILP should read the complete study.

The aggressive processor we consider has the following characteristics:

1. Up to 64 instruction issues and dispatches per clock with *no* issue restrictions, or 8 times the total issue width of the widest processor in 2016 (the IBM Power8) and with up to 32 times as many loads and stores allowed per clock! As we have discussed, there are serious complexity and power problems with large issue rates.
2. A tournament predictor with 1K entries and a 16-entry function return predictor. This predictor is comparable to the best predictors in 2016; the predictor is not a primary bottleneck. Mispredictions are handled in one cycle, but they limit the ability to speculate.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes.
4. Register renaming with 64 additional integer and 64 additional FP registers, which is somewhat less than the most aggressive processor in 2011. Because the study assumes a latency of only one cycle for all instructions (versus 15 or more on processors like the i7 or Power8), the effective number of rename registers is about five times larger than either of those processors.

Figure 3.45 shows the result for this configuration as we vary the window size. This configuration is more complex and expensive than existing implementations, especially in terms of the number of instruction issues. Nonetheless, it gives a useful upper limit on what future implementations might yield. The data in these figures are likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: for example, they may all be memory references. No one would even contemplate this capability in a processor for the near future. In addition, remember that in interpreting these results, cache misses and non-unit latencies were not taken into account, and both these effects have significant impacts.

The most startling observation in Figure 3.45 is that with the preceding realistic processor constraints, the effect of the window size for the integer programs is not as severe as for FP programs. This result points to the key difference between these two types of programs. The availability of loop-level parallelism in two of the FP programs means that the amount of ILP that can be exploited is higher, but for integer programs other factors—such as branch prediction, register renaming, and less parallelism, to start with—are all important limitations. This observation is critical because most of the market growth in the past decade—transaction processing, web servers, and the like—depended on integer performance, rather than floating point.

Wall’s study was not believed by some, but 10 years later, the reality had sunk in, and the combination of modest performance increases with significant hardware resources and major energy issues coming from incorrect speculation forced a change in direction. We will return to this discussion in our concluding remarks.

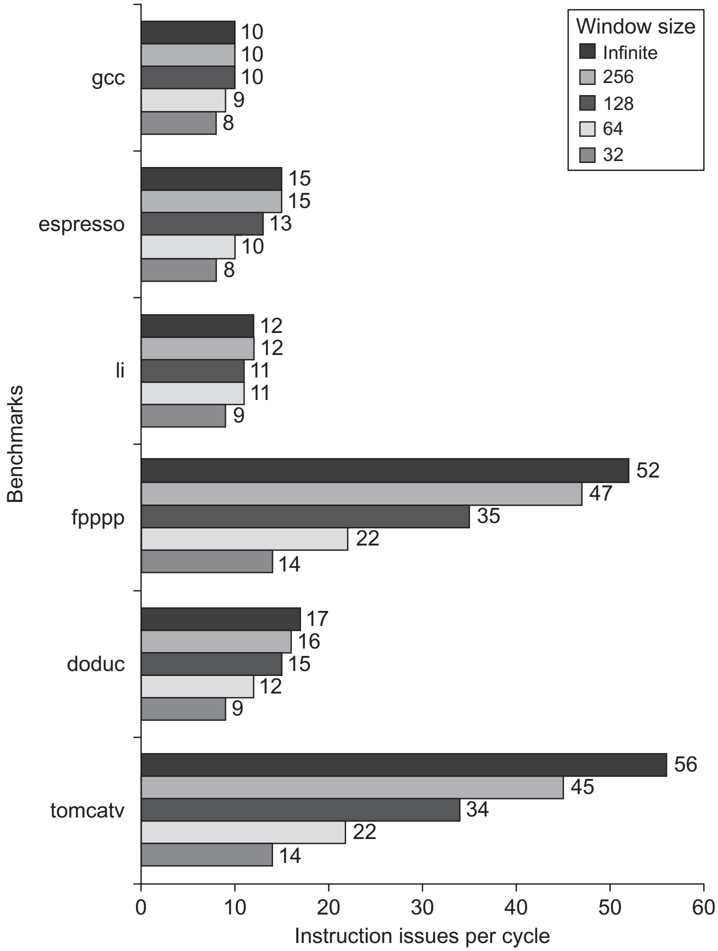


Figure 3.45 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have 1-cycle latency and that the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window.

3.14 Concluding Remarks: What's Ahead?

As 2000 began the focus on exploiting instruction-level parallelism was at its peak. In the first five years of the new century, it became clear that the ILP approach had likely peaked and that new approaches would be needed. By 2005 Intel and all the other major processor manufacturers had revamped their approach to focus on multicore. Higher performance would be achieved through thread-level parallelism rather than instruction-level parallelism, and the responsibility for using the processor efficiently would largely shift from the hardware to the software and the programmer. This change was the most significant change in processor architecture since the early days of pipelining and instruction-level parallelism some 25+ years earlier.

During the same period, designers began to explore the use of more data-level parallelism as another approach to obtaining performance. SIMD extensions enabled desktop and server microprocessors to achieve moderate performance increases for graphics and similar functions. More importantly, graphics processing units (GPUs) pursued aggressive use of SIMD, achieving significant performance advantages for applications with extensive data-level parallelism. For scientific applications, such approaches represent a viable alternative to the more general, but less efficient, thread-level parallelism exploited in multicores. The next chapter explores these developments in the use of data-level parallelism.

Many researchers predicted a major retrenchment in the use of ILP, predicting that two issue superscalar processors and larger numbers of cores would be the future. The advantages, however, of slightly higher issue rates and the ability of speculative dynamic scheduling to deal with unpredictable events, such as level-one cache misses, led to moderate ILP (typically about 4 issues/clock) being the primary building block in multicore designs. The addition of SMT and its effectiveness (both for performance and energy efficiency) further cemented the position of the moderate issue, out-of-order, speculative approaches. Indeed, even in the embedded market, the newest processors (e.g., the ARM Cortex-A9 and Cortex-A73) have introduced dynamic scheduling, speculation, and wider issues rates.

It is highly unlikely that future processors will try to increase the width of issue significantly. It is simply too inefficient from the viewpoint of silicon utilization and power efficiency. Consider the data in Figure 3.46 that show the five processors in the IBM Power series. Over more than a decade, there has been a modest improvement in the ILP support in the Power processors, but the dominant portion

	Power4	**Power5**	**Power6**	**Power7**	**Power8**
Introduced	2001	2004	2007	2010	2014
Initial clock rate (GHz)	1.3	1.9	4.7	3.6	3.3 GHz
Transistor count (M)	174	276	790	1200	4200
Issues per clock	5	5	7	6	8
Functional units per core	8	8	9	12	16
SMT threads per core	0	2	2	4	8
Cores/chip	2	2	2	8	12
SMT threads per core	0	2	2	4	8
Total on-chip cache (MiB)	1.5	2	4.1	32.3	103.0

Figure 3.46 Characteristics of five generations of IBM Power processors. All except the Power6, which is static and in-order, were dynamically scheduled; all the processors support two load/store pipelines. The Power6 has the same functional units as the Power5 except for a decimal unit. Power7 and Power8 use embedded DRAM for the L3 cache. Power9 has been described briefly; it further expands the caches and supports off-chip HBM.

of the increase in transistor count (a factor of more than 10 from the Power4 to the Power8) went to increasing the caches and the number of cores per die. Even the expansion in SMT support seems to be more of a focus than is an increase in the ILP throughput: The ILP structure from Power4 to Power8 went from 5 issues to 8, from 8 functional units to 16 (but not increasing from the original 2 load/store units), whereas the SMT support went from nonexistent to 8 threads/processor. A similar trend can be observed across the six generations of i7 processors, where almost all the additional silicon has gone to supporting more cores. The next two chapters focus on approaches that exploit data-level and thread-level parallelism.

3.15 Historical Perspective and References

Section M.5 (available online) features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics. Section M.5 covers both Chapter 3 and Appendix H.

Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell

Case Study: Exploring the Impact of Microarchitectural Techniques

Concepts illustrated by this case study

- Basic Instruction Scheduling, Reordering, Dispatch
- Multiple Issue and Hazards
- Register Renaming
- Out-of-Order and Speculative Execution
- Where to Spend Out-of-Order Resources

You are tasked with designing a new processor microarchitecture and you are trying to determine how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 3 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 3.47 provides a sequence of instructions and list of latencies.

3.1 [10] <3.1, 3.2> What is the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47 if no new instruction's execution could be

Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
fadd.d	+2
fmul.d	+4
fdiv.d	+10

```
Loop:       fld         f2,0(Rx)
I0:         fmul.d      f2,f0,f2
I1:         fdiv.d      f8,f2,f0
I2:         fld         f4,0(Ry)
I3:         fadd.d      f4,f0,f4
I4:         fadd.d      f10,f8,f2
I5:         fsd         f4,0(Ry)
I6:         addi        Rx,Rx,8
I7:         addi        Ry,Ry,8
I8:         sub         x20,x4,Rx
I9:         bnz         x20,Loop
```

Figure 3.47 Code and latencies for Exercises 3.1 through 3.6.

initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

3.2 [10] <3.1, 3.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.47 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with `<stall>` inserted where necessary to accommodate stated latencies. (*Hint:* an instruction with latency +2 requires two `<stall>` cycles to be inserted into the code sequence.) Think of it this

way: a one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + N has N extra stall cycles.

3.3 [15] <3.1, 3.2> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

3.4 [10] <3.1, 3.2> In the multiple-issue design of Exercise 3.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction $N+1$ begins execution in Execution Pipe 1 at the same time that instruction N begins in Pipe 0, and $N+1$ happens to require a shorter execution latency than N, then $N+1$ will complete before N (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 3.47 that demonstrate this hazard.

3.5 [20] <3.1, 3.2> Reorder the instructions to improve performance of the code in Figure 3.47. Assume the two-pipe machine in Exercise 3.3 and that the out-of-order completion issues of Exercise 3.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

3.6 [10/10/10] <3.1, 3.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.

a. [10] <3.1, 3.2> In your reordered code from Exercise 3.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

b. [10] <3.1, 3.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 3.5.

c. [10] <3.1, 3.2> What speedup did you obtain? (For this exercise, just color the $N+1$ iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)

3.7 [15] <3.4> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 3.48 shows example code that we would like our hardware to rename. A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them `T` registers, and assume that there are 64 of them, `T0` through `T63`) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the `src` (source) register designation, and the value in the table is the `T` register of the last destination that targeted that register. (Think of these table values as producers, and the `src` registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 3.48. Every time you see a destination register in the code, substitute the next available `T`, beginning with `T9`. Then update all the `src` registers accordingly, so that true data dependences are maintained. Show the resulting code. (*Hint:* see Figure 3.49.)

3.8 [20] <3.4> Exercise 3.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination `T` register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available `T` for it, but superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A SimpleScalar processor would therefore look up both `src` register mappings for each instruction and allocate a new `dest` mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any `dest`-to-`src` relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 3.50. Assume that we would like to simultaneously

```
Loop:          fld            f2,0(Rx)
I0:            fmul.d         f5,f0,f2
I1:            fdiv.d         f8,f0,f2
I2:            fld            f4,0(Ry)
I3:            fadd.d         f6,f0,f4
I4:            fadd.d         f10,f8,f2
I5:            sd             f4,0(Ry)
```

Figure 3.48 Sample code for register renaming practice.

```
I0:           fld             T9,0(Rx)
I1:           fmul.d          T10,F0,T9
...
```

Figure 3.49 Expected output of register renaming.

```
I0:          fmul.d          f5,f0,f2
I1:          fadd.d          f9,f5,f4
I2:          fadd.d          f5,f5,f2
I3:          fdiv.d          f2,f9,f0
```

Figure 3.50 Sample code for superscalar register renaming.

rename the first two instructions. Further assume that the next two available `T` registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups and then update the table per its destination's `T` register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that `T` register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 3.51 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code shown in Figure 3.50. Assume the table starts out with every entry equal to its index (`T0`=`0`; `T1`=`1`, …) (Figure 3.51).

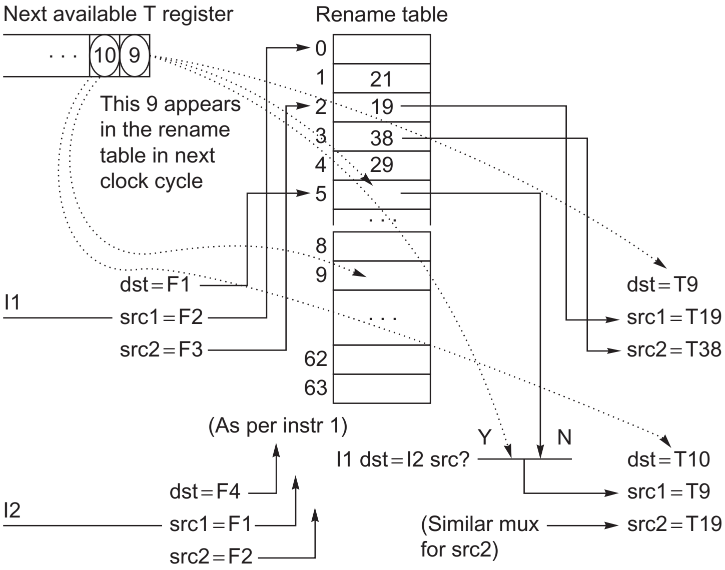

Next available T register
Rename table
. . .
10
9
This 9 appears in the rename table in next clock cycle
0
1
21
2
19
3
38
4
29
5
. . .
8
9
. . .
62
63
I1
dst = F1
src1 = F2
src2 = F3
dst = T9
src1 = T19
src2 = T38
(As per instr 1)
Y
N
I1 dst = I2 src?
dst = T10
src1 = T9
src2 = T19
I2
dst = F4
src1 = F1
src2 = F2
(Similar mux for src2)

Figure 3.51 Initial state of the register renaming table.

3.9 [5] <3.4> If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

```
addi x1, x1, x1
addi x1, x1, x1
addi x1, x1, x1
```

If the value of `x1` starts out as 5, what should its value be when this sequence has executed?

3.10 [20] <3.4, 3.7> Very long instruction word (VLIW) designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most *L* cycles later (where *L* is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 3.52. If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

3.11 [10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 3.53. All ops are one cycle except `LW` and `SW`, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?

b. [10] <3.3> Assume a static branch predictor, capable of recognizing a backward branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

```
Loop:   lw        x1,0(x2);    lw      x3,8(x2)
        <stall>
        <stall>
        addi      x10,x1,1;    addi    x11,x3,1
        sw        x1,0(x2);    sw      x3,8(x2)
        addi      x2,x2,8
        sub       x4,x3,x2
        bnz       x4,Loop
```

Figure 3.52 Sample VLIW code with two adds, two loads, and two stalls.

```
Loop:         lw x1,0(x2)
              addi        x1,x1, 1
              sw          x1,0(x2)
              addi        x2,x2,4
              sub         x4,x3,x2
              bnz         x4,Loop
```

Figure 3.53 Code loop for Exercise 3.11.

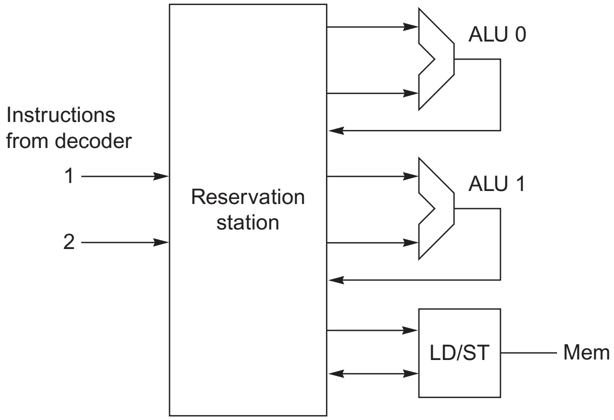


Figure 3.54 Microarchitecture for Exercise 3.12.

c. [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

3.12 [15/20/20/10/20] <3.4, 3.6> Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 3.54. Assume that the arithmetic-logical units (ALUs) can do all arithmetic ops (`fmul.d`, `fdiv.d`, `fadd.d`, `addi`, `sub`) and branches, and that the Reservation Station (RS) can dispatch, at most, one operation to each functional unit per cycle (one op to each ALU plus one memory op to the `fld`/ `fsd`).

a. [15] <3.4> Suppose all of the instructions from the sequence in Figure 3.47 are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. (*Hint:* look for read-after-write and write-after-write hazards. Assume the same functional unit latencies as in Figure 3.47.)

b. [20] <3.4> Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle *N*, with latencies as given in Figure 3.47. Show how the RS should dispatch these instructions out of order, clock by clock, to obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS before they're available for use—no bypassing.) How many clock cycles does the code sequence take?

c. [20] <3.4> Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0, the first two register-renamed instructions of this sequence appear in the RS. Assume it takes one clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 3.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?

d. [10] <3.4> If you wanted to improve the results of part (c), which would have helped most: (1) Another ALU? (2) Another LD/ST unit? (3) Full bypassing of ALU results to subsequent operations? or (4) Cutting the longest latency in half? What's the speedup?

e. [20] <3.6> Now let's consider speculation, the act of fetching, decoding, and executing beyond one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable). Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)

Exercises

3.13 [25] <3.7, 3.8> In this exercise, you will explore performance trade-offs between three processors that each employ different types of multithreading (MT). Each of these processors is superscalar, uses in-order pipelines, requires a fixed three-cycle stall following all loads and branches, and has identical L1 caches. Instructions from the same thread issued in the same cycle are read in program order and must not contain any data or control dependences.

- Processor A is a superscalar simultaneous MT architecture, capable of issuing up to two instructions per cycle from two threads.
- Processor B is a fine-grained MT architecture, capable of issuing up to four instructions per cycle from a single thread and switches threads on any pipeline stall.

- Processor C is a coarse-grained MT architecture, capable of issuing up to eight instructions per cycle from a single thread and switches threads on an L1 cache miss.

Our application is a list searcher, which scans a region of memory for a specific value stored in `R9` between the address range specified in `R16` and `R17`. It is parallelized by evenly dividing the search space into four equal-sized contiguous blocks and assigning one search thread to each block (yielding four threads). Most of each thread's runtime is spent in the following unrolled loop body:

```
loop: lw x1,0(x16)
      lw x2,8(x16)
      lw x3,16(x16)
      lw x4,24(x16)
      lw x5,32(x16)
      lw x6,40(x16)
      lw x7,48(x16)
      lw x8,56(x16)
      beq x9,x1,match0
      beq x9,x2,match1
      beq x9,x3,match2
      beq x9,x4,match3
      beq x9,x5,match4
      beq x9,x6,match5
      beq x9,x7,match6
      beq x9,x8,match7
      DADDIU x16,x16,#64
      blt x16,x17,loop
```

Assume the following:

- A barrier is used to ensure that all threads begin simultaneously.
- The first L1 cache miss occurs after two iterations of the loop.
- None of the BEQAL branches is taken.
- The BLT is always taken.
- All three processors schedule threads in a round-robin fashion.

Determine how many cycles are required for each processor to complete the first two iterations of the loop.

3.14 [25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision *aX* plus *Y*) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, $Y = aX + Y$, for a vector length 100. Initially, `R1` is set to the base address of array *X* and `R2` is set to the base address of *Y*:

```
addi  x4,x1,#800 ; x1 = upper bound for X
```

```
foo: fld      F2,0(x1)   ; (F2) = X(i)
     fmul.d   F4,F2,F0   ; (F4) = a*X(i)
     fld      F6,0(x2)   ; (F6) = Y(i)
     fadd.d   F6,F4,F6   ; (F6) = a*X(i) + Y(i)
     fsd      F6,0(x2)   ; Y(i) = a*X(i) + Y(i)
     addi     x1,x1,#8   ; increment X index
     addi     x2,x2,#8   ; increment Y index
     sltu     x3,x1,x4   ; test: continue loop?
     bnez     x3,foo     ; loop if needed
```

Assume the functional unit latencies as shown in the following table. Assume a one-cycle delayed branch that resolves in the ID stage. Assume that results are fully bypassed.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	4
Integer operations and all loads	Any	2

a. [25] <3.2> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

b. [25] <3.2> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

c. [25] <3.7> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 3.20. We will compare two degrees of loop unrolling. First, unroll the loop 6 times to extract ILP and schedule it without any stalls (i.e., completely empty issue cycles), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 10 times. Ignore the branch delay slot. Show the two schedules. What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

3.15 [20/20] <3.4, 3.5, 3.7, 3.8> In this exercise, we will look at how variations on Tomasulo's algorithm perform when running the loop from Exercise 3.14. The functional units (FUs) are described in the following table.

FU type	Cycles in EX	Number of FUs	Number of reservation stations
Integer	1	1	5
FP adder	10	1	3
FP multiplier	15	1	2

Assume the following:

- Functional units are not pipelined.
- There is no forwarding between functional units; results are communicated by the common data bus (CDB).
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus, the pipeline is IF/ID/IS/EX/WB.
- Loads require one clock cycle.
- The issue (IS) and write-back (WB) result stages each require one clock cycle.
- There are five load buffer slots and five store buffer slots.
- Assume that the Branch on Not Equal to Zero (BNEZ) instruction requires one clock cycle.

a. [20] <3.4–3.5> For this problem use the single-issue Tomasulo MIPS pipeline of Figure 3.10 with the pipeline latencies from the preceding table. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop. How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:
   - Iteration (loop iteration number)
   - Instruction
   - Issues (cycle when instruction issues)
   - Executes (cycle when instruction executes)
   - Memory access (cycle when memory is accessed)
   - Write CDB (cycle when result is written to the CDB)
   - Comment (description of any event on which the instruction is waiting)

   Show three iterations of the loop in your table. You may ignore the first instruction.

b. [20] <3.7, 3.8> Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).

3.16 [10] <3.4> Tomasulo's algorithm has a disadvantage: only one result can compute per clock per CDB. Use the hardware configuration and latencies from the previous question and find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

3.17 [20] <3.3> An (*m*,*n*) correlating branch predictor uses the behavior of the most recent *m* executed branches to choose from 2^m predictors, each of which is an *n*-bit predictor. A two-level local predictor works in a similar fashion, but only keeps track of the past behavior of each individual branch to predict future behavior.

There is a design trade-off involved with such predictors: correlating predictors require little memory for history, which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions. For this exercise, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:

Correlating predictor

Entry	Branch	Last outcome	Prediction
0	0	T	T with one misprediction
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT with one misprediction
7	3	NT	NT

Local predictor

Entry	Branch	Last 2 outcomes (right is most recent)	Prediction
0	0	T,T	T with one misprediction
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T with one misprediction
6	1	NT,T	NT
7	1	NT,NT	NT

Branch PC (word address)	Outcome
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3.18 [10] <3.9> Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.

3.19 [10/5] <3.9> Consider a branch-target buffer that has penalties of zero, two, and two clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch.

a. [10] <3.9> What is the penalty in clock cycles when an unconditional branch is found in the buffer?

b. [10] <3.9> Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a two-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?

This page intentionally left blank

4.1 Introduction 282
4.2 Vector Architecture 283
4.3 SIMD Instruction Set Extensions for Multimedia 304
4.4 Graphics Processing Units 310
4.5 Detecting and Enhancing Loop-Level Parallelism 336
4.6 Cross-Cutting Issues 345
4.7 Putting It All Together: Embedded Versus Server GPUs and Tesla Versus Core i7 346
4.8 Fallacies and Pitfalls 353
4.9 Concluding Remarks 357
4.10 Historical Perspective and References 357
Case Study and Exercises by Jason D. Bakos 357

4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

We call these algorithms *data parallel* algorithms because their parallelism comes from simultaneous operations across large sets of data rather than from multiple threads of control.

W. Daniel Hillis and Guy L. Steele,
"Data parallel algorithms," *Commun. ACM* (1986)

If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?

Seymour Cray, Father of the Supercomputer
(arguing for two powerful vector processors versus many simple processors)

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00004-3
© 2019 Elsevier Inc. All rights reserved.

4.1 Introduction

A question for the single instruction multiple data (SIMD) architecture, which Chapter 1 introduced, has always been just how wide a set of applications has significant data-level parallelism (DLP). Five years after the SIMD classification was proposed (Flynn, 1966), the answer is not only the matrix-oriented computations of scientific computing but also the media-oriented image and sound processing and machine learning algorithms, as we will see in Chapter 7. Since a multiple instruction multiple data (MIMD) architecture needs to fetch one instruction per data operation, single instruction multiple data (SIMD) is potentially more energy-efficient since a single instruction can launch many data operations. These two answers make SIMD attractive for personal mobile devices as well as for servers. Finally, perhaps the biggest advantage of SIMD versus MIMD is that the programmer continues to think sequentially yet achieves parallel speedup by having parallel data operations.

This chapter covers three variations of SIMD: vector architectures, multimedia SIMD instruction set extensions, and graphics processing units (GPUs).[1]

The first variation, which predates the other two by more than 30 years, extends pipelined execution of many data operations. These *vector architectures* are easier to understand and to compile to than other SIMD variations, but they were considered too expensive for microprocessors until recently. Part of that expense was in transistors, and part was in the cost of sufficient dynamic random access memory (DRAM) bandwidth, given the widespread reliance on caches to meet memory performance demands on conventional microprocessors.

The second SIMD variation borrows from the SIMD name to mean basically simultaneous parallel data operations and is now found in most instruction set architectures that support multimedia applications. For x86 architectures, the SIMD instruction extensions started with the MMX (multimedia extensions) in 1996, which were followed by several SSE (streaming SIMD extensions) versions in the next decade, and they continue until this day with AVX (advanced vector extensions). To get the highest computation rate from an x86 computer, you often need to use these SIMD instructions, especially for floating-point programs.

The third variation on SIMD comes from the graphics accelerator community, offering higher potential performance than is found in traditional multicore computers today. Although GPUs share features with vector architectures, they have their own distinguishing characteristics, in part because of the ecosystem in which they evolved. This environment has a system processor and system memory in addition to the GPU and its graphics memory. In fact, to recognize those distinctions, the GPU community refers to this type of architecture as *heterogeneous*.

---

[1] This chapter is based on material in Appendix F, “Vector Processors,” by Krste Asanovic, and Appendix G, “Hardware and Software for VLIW and EPIC” from the 5th edition of this book; on material in Appendix A, “Graphics and Computing GPUs,” by John Nickolls and David Kirk, from the 5th edition of *Computer Organization and Design*; and to a lesser extent on material in “Embracing and Extending 20th-Century Instruction Set Architectures,” by Joe Gebis and David Patterson, *IEEE Computer*, April 2007.

For problems with lots of data parallelism, all three SIMD variations share the advantage of being easier on the programmer than classic parallel MIMD programming.

The goal of this chapter is for architects to understand why vector is more general than multimedia SIMD, as well as the similarities and differences between vector and GPU architectures. Because vector architectures are supersets of the multimedia SIMD instructions, including a better model for compilation, and because GPUs share several similarities with vector architectures, we start with vector architectures to set the foundation for the following two sections. The next section introduces vector architectures, and Appendix G goes much deeper into the subject.

4.2 Vector Architecture

> The most efficient way to execute a vectorizable application is a vector processor.
>
> **Jim Smith,**
> *International Symposium on Computer Architecture* (1994)

Vector architectures grab sets of data elements scattered about memory, place them into large sequential register files, operate on data in those register files, and then disperse the results back into memory. A single instruction works on vectors of data, which results in dozens of register-register operations on independent data elements.

These large register files act as compiler-controlled buffers, both to hide memory latency and to leverage memory bandwidth. Because vector loads and stores are deeply pipelined, the program pays the long memory latency only once per vector load or store versus once per element, thus amortizing the latency over, say, 32 elements. Indeed, vector programs strive to keep the memory busy.

The power wall leads architects to value architectures that can deliver good performance without the energy and design complexity costs of highly out-of-order superscalar processors. Vector instructions are a natural match to this trend because architects can use them to increase performance of simple in-order scalar processors without greatly raising energy demands and design complexity. In practice, developers can express many of the programs that ran well on complex out-of-order designs more efficiently as data-level parallelism in the form of vector instructions, as Kozyrakis and Patterson (2002) showed.

RV64V Extension

We begin with a vector processor consisting of the primary components that Figure 4.1 shows. It is loosely based on the 40-year-old Cray-1, which was one of the first supercomputers. At the time of the writing of this edition, the RISC-V vector instruction set extension RVV was still under development. (The vector extension by itself is called RVV, so RV64V refers to the RISC-V base instructions

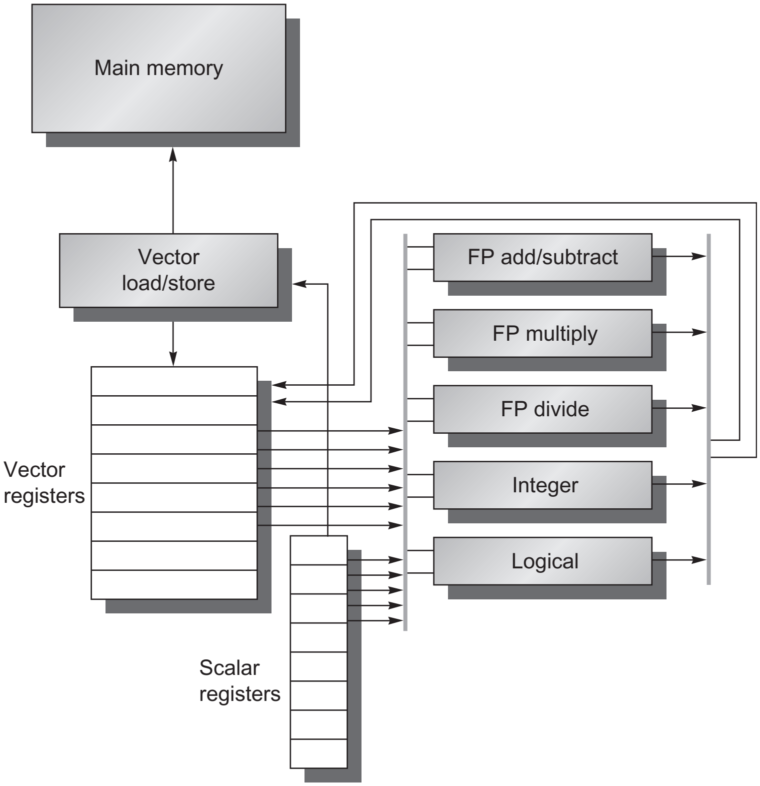


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes a RISC-V scalar architecture. There are also 32 vector registers, and all the functional units are vector functional units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches *(thick gray lines)* connects these ports to the inputs and outputs of the vector functional units.

plus the vector extension.) We show a subset of RV64V, trying to capture its essence in a few pages.

The primary components of the instruction set architecture of RV64V are the following:

- *Vector registers*—Each vector register holds a single vector, and RV64V has 32 of them, each 64 bits wide. The vector register file needs to provide enough ports to feed all the vector functional units. These ports will allow a high degree of overlap among vector operations to different vector registers. The read and write ports, which total at least 16 read ports and 8 write ports, are connected to the functional unit inputs or outputs by a pair of crossbar switches. One way to increase the register file bandwidth is to compose it from multiple banks, which work well with relatively long vectors.

- *Vector functional units*—Each unit is fully pipelined in our implementation, and it can start a new operation on every clock cycle. A control unit is needed to detect hazards, both structural hazards for functional units and data hazards on register accesses. Figure 4.1 shows that we assume an implementation of RV64V has five functional units. For simplicity, we focus on the floating-point functional units in this section.
- *Vector load/store unit*—The vector memory unit loads or stores a vector to or from memory. The vector loads and stores are fully pipelined in our hypothetical RV64V implementation so that words can be moved between the vector registers and memory with a bandwidth of one word per clock cycle, after an initial latency. This unit would also normally handle scalar loads and stores.
- *A set of scalar registers*—Scalar registers can likewise provide data as input to the vector functional units, as well as compute addresses to pass to the vector load/store unit. These are the normal 31 general-purpose registers and 32 floating-point registers of RV64G. One input of the vector functional units latches scalar values as they are read out of the scalar register file.

Figure 4.2 lists the RV64V vector instructions we use in this section. The description in Figure 4.2 assumes that the input operands are all vector registers, but there are also versions of these instructions where an operand can be a scalar register (`xi` or `fi`). RV64V uses the suffix `.vv` when both are vectors, `.vs` when the second operand is a scalar, and `.sv` when the first is a scalar register. Thus these three are all valid RV64V instructions: `vsub.vv`, `vsub.vs`, and `vsub.sv`. (Add and other commutative operations have only the first two versions, as `vadd.sv` and `vadd.sv` would be redundant.) Because the operands determine the version of the instruction, we usually let the assembler supply the appropriate suffix. The vector functional unit gets a copy of the scalar value at instruction issue time.

Although the traditional vector architectures didn't support narrow data types efficiently, vectors naturally accommodate varying data sizes (Kozyrakis and Patterson, 2002). Thus, if a vector register has 32 64-bit elements, then 128 × 16-bit elements, and even 256 × 8-bit elements are equally valid views. Such hardware multiplicity is why a vector architecture can be useful for multimedia applications as well as for scientific applications.

Note that the RV64V instructions in Figure 4.2 omit the data type and size! An innovation of RV64V is to associate a data type and data size *with each vector register*, rather than the normal approach of the instruction supplying that information. Thus, before executing the vector instructions, a program configures the vector registers being used to specify their data type and widths. Figure 4.3 lists the options for RV64V.

Mnemonic	Name	Description
`vadd`	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
`vsub`	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
`vmul`	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
`vdiv`	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
`vrem`	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
`vsqrt`	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
`vsll`	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
`vsrl`	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
`vsra`	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
`vxor`	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
`vor`	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
`vand`	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
`vsgnj`	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
`vsgnjn`	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
`vsgnjx`	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
`vld`	Load	Load vector register V[rd] from memory starting at address R[rs1]
`vlds`	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., $R[rs1]+i\times R[rs2]$)
`vldx`	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at $R[rs2]+V[rs2]$ (i.e., V[rs2] is an index)
`vst`	Store	Store vector register V[rd] into memory starting at address R[rs1]
`vsts`	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., $R[rs1]+i\times R[rs2]$)
`vstx`	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at $R[rs2]+V[rs2]$ (i.e., V[rs2] is an index)
`vpeq`	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
`vpne`	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
`vplt`	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
`vpxor`	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
`vpor`	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
`vpand`	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
`setvl`	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

Figure 4.2 The RV64V vector instructions. All use the R instruction format. Each vector operation with two operands is shown with both operands being vector (`.vv`), but there are also versions where the second operand is a scalar register (`.vs`) and, when it makes a difference, where the first operand is a scalar register and the second is a vector register (`.sv`). The type and width of the operands are determined by configuring each vector register rather than being supplied by the instruction. In addition to the vector registers and predicate registers, there are two vector control and status registers (CSRs), `vl` and `vctype`, discussed below. The strided and indexed data transfers are also explained later. Once completed, RV64 will surely have more instructions, but the ones in this figure will be included.

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

Figure 4.3 Data sizes supported for RV64V assuming it also has the single- and double-precision floating-point extensions RVS and RVD. Adding RVV to such a RISC-V design means the scalar unit must also add RVH, which is a scalar instruction extension to support half-precision (16-bit) IEEE 754 floating point. Because RV32V would not have doubleword scalar operations, it could drop 64-bit integers from the vector unit. If a RISC-V implementation didn't include RVS or RVD, it could omit the vector floating-point instructions.

One reason for *dynamic register typing* is that many instructions are required for a conventional vector architecture that supports such variety. Given the combinations of data types and sizes in Figure 4.3, if not for dynamic register typing, Figure 4.2 would be several pages long!

Dynamic typing also lets programs disable unused vector registers. As a consequence, enabled vector registers are allocated all the vector memory as long vectors. For example, assume we have 1024 bytes of vector memory, if 4 vector registers are enabled and they are type 64–bit floats, the processor would give each vector register 256 bytes or 256/8=32 elements. This valiue is called the maximum vector length (mvl), which is set by the processor and cannot be changed by sofware.

One complaint about vector architectures is that their larger state means slower context switch time. Our implementation of RV64V increases state a factor of 3: from $2 \times 32 \times 8 = 512$ bytes to $2 \times 32 \times 1024 = 1536$ bytes. A pleasant side effect of dynamic register typing is that the program can configure vector registers as *disabled* when they are not being used, so there is no need to save and restore them on a context switch.

A third benefit of dynamic register typing is that conversions between different size operands can be implicit depending on the configuration of the registers rather than as additional explicit conversion instructions. We'll see an example of this benefit in the next section.

The names `vld` and `vst` denote vector load and vector store, and they load or store an entire vectors of data. One operand is the vector register to be loaded or stored; the other operand, which is a RV64G general-purpose register, is the starting address of the vector in memory. Vector needs more registers beyond the vector registers themselves. The vector-length register `vl` is used when the natural vector length is not equal to mvl, the vector-type register `vctype` records register types, and the predicate registers `p`$_i$ are used when loops involve IF statements. We'll see them in action in the following example.

With a vector instruction, the system can perform the operations on the vector data elements in many ways, including operating on many elements simultaneously. This flexibility lets vector designs use slow but wide execution units to achieve high performance at low power. Furthermore, the independence of elements within a vector instruction set allows scaling of functional units without performing additional costly dependency checks, as superscalar processors require.

How Vector Processors Work: An Example

We can best understand a vector processor by looking at a vector loop for RV64V. Let's take a typical vector problem, which we use throughout this section:

`Y = a × X + Y`

`X` and `Y` are vectors, initially resident in memory, and `a` is a scalar. This problem is the *SAXPY* or *DAXPY* loop that forms the inner loop of the Linpack benchmark (Dongarra et al., 2003). (SAXPY stands for single-precision a × X plus Y, and DAXPY for double precision a × X plus Y.) Linpack is a collection of linear algebra routines, and the Linpack benchmark consists of routines for performing Gaussian elimination.

For now, let us assume that the number of elements, or *length*, of a vector register (32) matches the length of the vector operation we are interested in. (This restriction will be lifted shortly.)

Example Show the code for RV64G and RV64V for the DAXPY loop. For this example, assume that X and Y have 32 elements and the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively. (A subsequent example covers when they do not have 32 elements.)

Answer Here is the RISC-V code:

```
      fld    f0,a        # Load scalar a
      addi   x28,x5,#256 # Last address to load
Loop: fld    f1,0(x5)    # Load X[i]
      fmul.d f1,f1,f0    # a × X[i]
      fld    f2,0(x6)    # Load Y[i]
      fadd.d f2,f2,f1    # a × X[i] + Y[i]
      fsd    f2,0(x6)    # Store into Y[i]
      addi   x5,x5,#8    # Increment index to X
      addi   x6,x6,#8    # Increment index to Y
      bne    x28,x5,Loop # Check if done
```

Here is the RV64V code for DAXPY:

```
vsetdcfg  4*FP64      # Enable 4 DP FP vregs
fld       f0,a        # Load scalar a
vld       v0,x5       # Load vector X
vmul      v1,v0,f0    # Vector-scalar mult
vld       v2,x6       # Load vector Y
vadd      v3,v1,v2    # Vector-vector add
vst       v3,x6       # Store the sum
vdisable              # Disable vector regs
```

Note that the assembler determines which version of the vector operations to generate. Because the multiply has a scalar operand, it generates `vmul.vs`, whereas the add doesn't, so it generates `vadd.vv`.

The initial instruction configures the first four vector registers to hold 64-bit floating-point data. The last instruction disables all vector registers. If a context switch happened after the last instruction, there is no additional state to save.

The most dramatic difference between the preceding scalar and vector code is that the vector processor greatly reduces the dynamic instruction bandwidth, executing only 8 instructions versus 258 for RV64G. This reduction occurs because the vector operations work on 32 elements and the overhead instructions that constitute nearly half the loop on RV64G are not present in the RV64V code. When the compiler produces vector instructions for such a sequence, and the resulting code spends much of its time running in vector mode, the code is said to be *vectorized* or *vectorizable*. Loops can be vectorized when they do not have dependences between iterations of a loop, which are called *loop-carried dependences* (see Section 4.5).

Another important difference between RV64G and RV64V is the frequency of pipeline interlocks for a simple implementation of RV64G. In the straightforward RV64G code, every `fadd.d` must wait for a `fmul.d`, and every `fsd` must wait for the `fadd.d`. On the vector processor, each vector instruction will stall only for the first element in each vector, and then subsequent elements will flow smoothly down the pipeline. Thus pipeline stalls are required only once per vector *instruction*, rather than once per vector *element*. Vector architects call forwarding of element-dependent operations *chaining*, in that the dependent operations are "chained" together. In this example, the pipeline stall frequency on RV64G will be about 32 × higher than it is on RV64V. Software pipelining, loop unrolling (Appendix H), or out-of-order execution can reduce the pipeline stalls on RV64G; however, the large difference in instruction bandwidth cannot be reduced substantially.

Let's show off the dynamic register typing before discussing performance of the code.

Example A common use of multiply-accumulate operations is to multiply using narrow data and to accumulate at a wider size to increase the accuracy of a sum of products. Show how the preceding code would change if `X` and `a` were single-precision instead of a double-precision floating point. Next, show the changes to this code if we switch `X`, `Y`, and `a` from floating-point type to integers.

Answer The changes are underlined in the following code. Amazingly, the same code works with two small changes: the configuration instruction includes one single-precision vector, and the scalar load is now single-precision:

```
vsetdcfg 1*FP32,3*FP64  # 1 32b, 3 64b vregs
flw      f0,a           # Load scalar a
vld      v0,x5          # Load vector X
vmul     v1,v0,f0       # Vector-scalar mult
vld      v2,x6          # Load vector Y
vadd     v3,v1,v2       # Vector-vector add
vst      v3,x6          # Store the sum
vdisable                # Disable vector regs
```

Note that RV64V hardware will implicitly perform a conversion from the narrower single-precision to the wider double-precision in this setup.

The switch to integers is almost as easy, but we must now use an integer load instruction and integer register to hold the scalar value:

```
vsetdcfg 1*X32,3*X64  # 1 32b, 3 64b int reg
lw       x7,a         # Load scalar a
vld      v0,x5        # Load vector X
vmul     v1,v0,x7     # Vector-scalar mult
vld      v2,x6        # Load vector Y
vadd     v3,v1,v2     # Vector-vector add
vst      v3,x6        # Store the sum
vdisable              # Disable vector regs
```

Vector Execution Time

The execution time of a sequence of vector operations primarily depends on three factors: (1) the length of the operand vectors, (2) structural hazards among the operations, and (3) the data dependences. Given the vector length and the *initiation rate*, which is the rate at which a vector unit consumes new operands and produces new results, we can compute the time for a single vector instruction.

All modern vector computers have vector functional units with multiple parallel pipelines (or *lanes*) that can produce two or more results per clock cycle, but they may also have some functional units that are not fully pipelined. For simplicity, our RV64V implementation has one lane with an initiation rate of one element per clock cycle for individual operations. Thus the execution time in clock cycles for a single vector instruction is approximately the vector length.

To simplify the discussion of vector execution and vector performance, we use the notion of a *convoy*, which is the set of vector instructions that could potentially execute together. The instructions in a convoy *must not* contain any structural hazards; if such hazards were present, the instructions would need to be serialized and initiated in different convoys. Thus the `vld` and the following `vmul` in the preceding example can be in the same convoy. As we will soon see, you can estimate performance of a section of code by counting the number of convoys. To keep this analysis simple, we assume that a convoy of instructions must complete execution before any other instructions (scalar or vector) can begin execution.

It might seem that in addition to vector instruction sequences with structural hazards, sequences with read-after-write dependency hazards should also be in separate convoys. However, chaining allows them to be in the same convoy since it allows a vector operation to start as soon as the individual elements of its vector source operand become available: the results from the first functional unit in the chain are “forwarded” to the second functional unit. In practice, we often implement chaining by allowing the processor to read and write a particular vector register at the same time, albeit to different elements. Early implementations of chaining worked just like forwarding in scalar pipelining, but this restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, assuming that we don't generate a structural hazard. All modern vector architectures support flexible chaining, which we assume throughout this chapter.

To turn convoys into execution time, we need a metric to estimate the length of a convoy. It is called a *chime*, which is simply the unit of time taken to execute one convoy. Thus a vector sequence that consists of m convoys executes in m chimes; for a vector length of n, for our simple RV64V implementation, this is approximately $m \times n$ clock cycles.

The chime approximation ignores some processor-specific overheads, many of which are dependent on vector length. Therefore measuring time in chimes is a better approximation for long vectors than for short ones. We will use the chime measurement, rather than clock cycles per result, to indicate explicitly that we are ignoring certain overheads.

If we know the number of convoys in a vector sequence, we know the execution time in chimes. One source of overhead ignored in measuring chimes is any limitation on initiating multiple vector instructions in a single clock cycle. If only one vector instruction can be initiated in a clock cycle (the reality in most vector processors), the chime count will underestimate the actual execution time of a convoy. Because the length of vectors is typically much greater than the number of instructions in the convoy, we will simply assume that the convoy executes in one chime.

Example Show how the following code sequence lays out in convoys, assuming a single copy of each vector functional unit:

```
vld   v0,x5        # Load vector X
vmul  v1,v0,f0     # Vector-scalar multiply
vld   v2,x6        # Load vector Y
vadd  v3,v1,v2     # Vector-vector add
vst   v3,x6        # Store the sum
```

How many chimes will this vector sequence take? How many cycles per FLOP (floating-point operation) are needed, ignoring vector instruction issue overhead?

Answer The first convoy starts with the first `vld` instruction. The `vmul` is dependent on the first `vld`, but chaining allows it to be in the same convoy.

The second `vld` instruction must be in a separate convoy because there is a structural hazard on the load/store unit for the prior `vld` instruction. The `vadd` is dependent on the second `vld`, but it can again be in the same convoy via chaining. Finally, the `vst` has a structural hazard on the `vld` in the second convoy, so it must go in the third convoy. This analysis leads to the following layout of vector instructions into convoys:

1. `vld` `vmul`
2. `vld` `vadd`
3. `vst`

The sequence requires three convoys. Because the sequence takes three chimes and there are two floating-point operations per result, the number of cycles per FLOP is 1.5 (ignoring any vector instruction issue overhead). Note that, although we allow the `vld` and `vmul` both to execute in the first convoy, most vector machines will take 2 clock cycles to initiate the instructions.

This example shows that the chime approximation is reasonably accurate for long vectors. For example, for 32-element vectors, the time in chimes is 3, so the sequence would take about 32 × 3 or 96 clock cycles. The overhead of issuing convoys in two separate clock cycles would be small.

Another source of overhead is far more significant than the issue limitation. The most important source of overhead ignored by the chime model is vector *start-up time*, which is the latency in clock cycles until the pipeline is full. The start-up time is principally determined by the pipelining latency of the vector functional unit. For RV64V, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for vector loads. All functional units are fully pipelined. The pipeline depths are 6 clock cycles for floating-point add, 7 for floating-point multiply, 20 for floating-point divide, and 12 for vector load.

Given these vector basics, the next several sections will give optimizations that either improve the performance or increase the types of programs that can run well on vector architectures. In particular, they will answer these questions:

- How can a vector processor execute a single vector faster than one element per clock cycle? Multiple elements per clock cycle improve performance.
- How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length (mvl)? Because most application vectors don't match the architecture vector length, we need an efficient solution to this common case.
- What happens when there is an IF statement inside the code to be vectorized? More code can vectorize if we can efficiently handle conditional statements.
- What does a vector processor need from the memory system? Without sufficient memory bandwidth, vector execution can be futile.
- How does a vector processor handle multiple dimensional matrices? This popular data structure must vectorize for vector architectures to do well.
- How does a vector processor handle sparse matrices? This popular data structure must vectorize also.

- How do you program a vector computer? Architectural innovations that are a mismatch to programming languages and their compilers may not get widespread use.

The rest of this section introduces each of these optimizations of the vector architecture, and Appendix G goes into greater depth.

Multiple Lanes: Beyond One Element per Clock Cycle

A critical advantage of a vector instruction set is that it allows software to pass a large amount of parallel work to hardware using only a single short instruction. One vector instruction can include scores of independent operations yet be encoded in the same number of bits as a conventional scalar instruction. The parallel semantics of a vector instruction allow an implementation to execute these elemental operations using a deeply pipelined functional unit, as in the RV64V implementation we've studied so far; an array of parallel functional units; or a combination of parallel and pipelined functional units. Figure 4.4 illustrates how to improve vector performance by using parallel pipelines to execute a vector add instruction.

The RV64V instruction set has the property that all vector arithmetic instructions only allow element *N* of one vector register to take part in operations with element *N* from other vector registers. This dramatically simplifies the design of a highly parallel vector unit, which can be structured as multiple parallel *lanes*. As with a traffic highway, we can increase the peak throughput of a vector unit by adding more lanes. Figure 4.5 shows the structure of a four-lane vector unit. Thus going to four lanes from one lane reduces the number of clocks for a chime from 32 to 8. For multiple lanes to be advantageous, both the applications and the architecture must support long vectors; otherwise, they will execute so quickly that you'll run out of instruction bandwidth, requiring ILP techniques (see Chapter 3) to supply enough vector instructions.

Each lane contains one portion of the vector register file and one execution pipeline from each vector functional unit. Each vector functional unit executes vector instructions at the rate of one element group per cycle using multiple pipelines, one per lane. The first lane holds the first element (element 0) for all vector registers, and so the first element in any vector instruction will have its source and destination operands located in the first lane. This allocation allows the arithmetic pipeline local to the lane to complete the operation without communicating with other lanes. Avoiding interlane communication reduces the wiring cost and register file ports required to build a highly parallel execution unit and helps explain why vector computers can complete up to 64 operations per clock cycle (2 arithmetic units and 2 load/store units across 16 lanes).

Adding multiple lanes is a popular technique to improve vector performance as it requires little increase in control complexity and does not require changes to existing machine code. It also allows designers to trade off die area, clock rate, voltage, and energy without sacrificing peak performance. If the clock rate of a

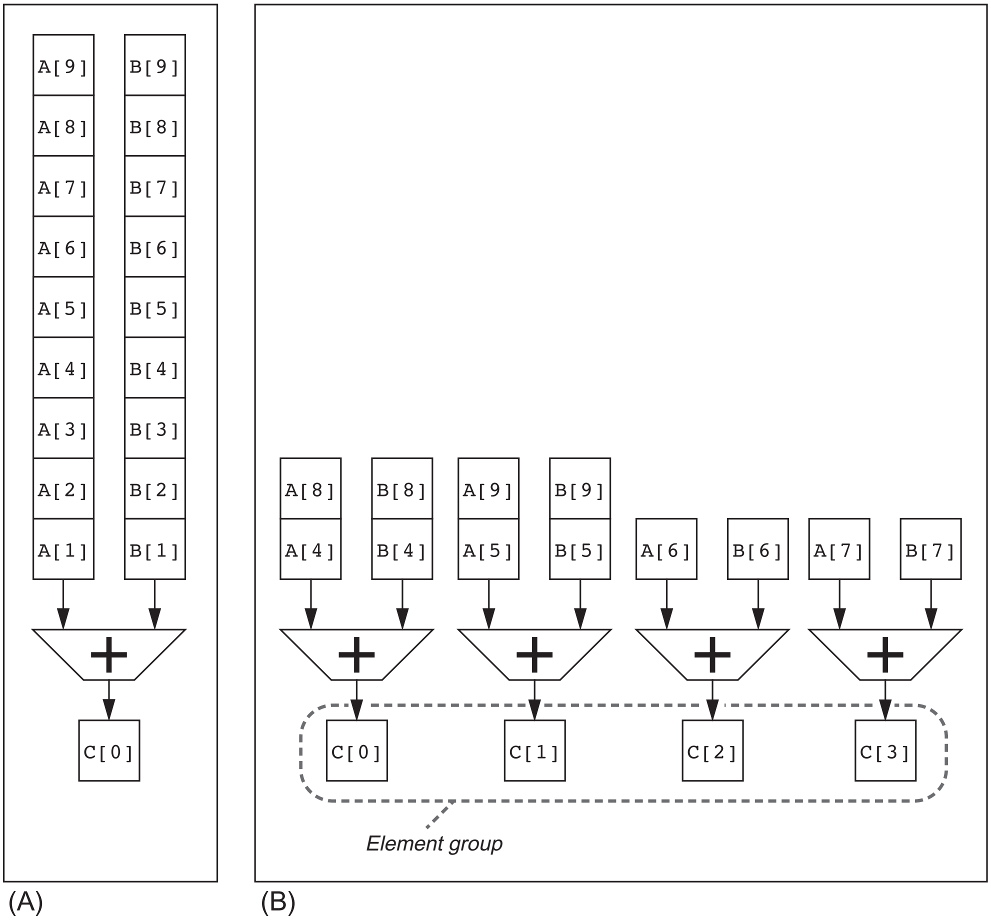


Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The vector processor (A) on the left has a single add pipeline and can complete one addition per clock cycle. The vector processor (B) on the right has four add pipelines and can complete four additions per clock cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. Reproduced with permission from Asanovic, K., 1998. Vector Microprocessors (Ph.D. thesis). Computer Science Division, University of California, Berkeley.

vector processor is halved, doubling the number of lanes will retain the same peak performance.

Vector-Length Registers: Handling Loops Not Equal to 32

A vector register processor has a natural vector length determined by the maximum vector length (mvl). This length, which was 32 in our example above, is unlikely

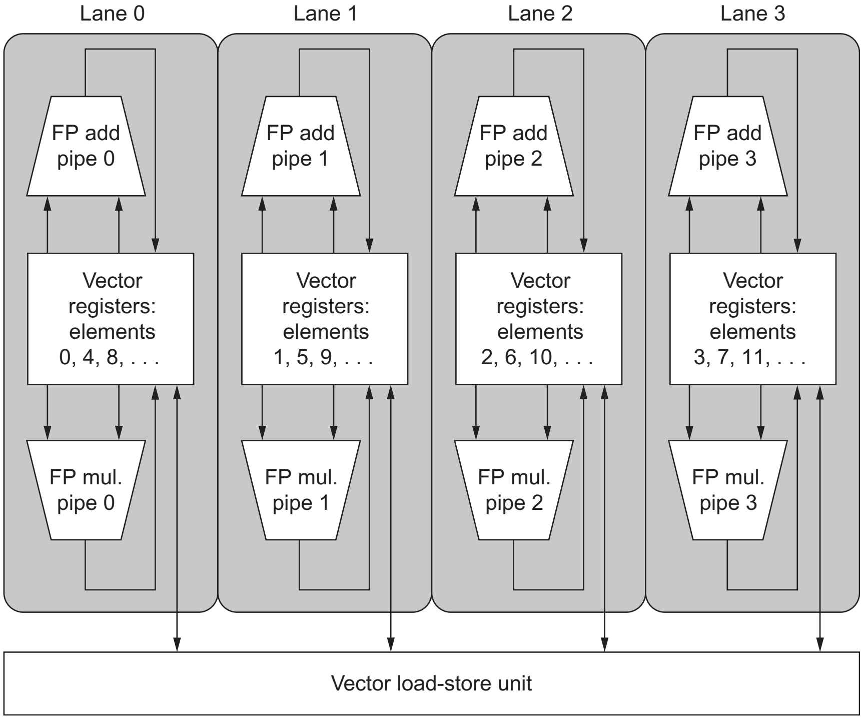

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

Figure 4.5 Structure of a vector unit containing four lanes. The vector register memory is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction. Note how each section of the vector register file needs to provide only enough ports for pipelines local to its lane. This figure does not show the path to provide the scalar operand for vector-scalar instructions, but the scalar processor (or Control Processor) broadcasts a scalar value to all lanes.

to match the real vector length in a program. Moreover, in a real program, the length of a particular vector operation is often *unknown* at compile time. In fact, a single piece of code may require different vector lengths. For example, consider this code:

```
for (i=0; i <n; i=i+1)
   Y[i] = a * X[i] + Y[i];
```

The size of all the vector operations depends on n, which may not even be known until run time. The value of n might also be a parameter to a procedure containing the preceding loop and therefore subject to change during execution.

The solution to these problems is to add a *vector-length register* (`vl`). The `vl` controls the length of any vector operation, including a vector load or store. The value in the `vl`, however, cannot be greater than the maximum vector length (mvl). This solves our problem as long as the real length is less than or equal to the maximum vector length (`mvl`). This parameter means the length of vector registers can grow in later computer generations without changing the instruction set. As we will see in the next section, multimedia SIMD extensions have no equivalent of `mvl`, so they expand the instruction set every time they increase their vector length.

What if the value of `n` is not known at compile time and thus may be greater than the `mvl`? To tackle the second problem where the vector is longer than the maximum length, a technique called *strip mining* is traditionally used. Strip mining is the generation of code such that each vector operation is done for a size less than or equal to the `mvl`. One loop handles any number of iterations that is a multiple of the `mvl` and another loop that handles any remaining iterations and must be less than the `mvl`. RISC-V has a better solution than a separate loop for strip mining. The instruction `setvl` writes the smaller of the `mvl` and the loop variable `n` into `vl` (and to another register). If the number of iterations of the loop is larger than `n`, then the fastest the loop can compute is `mvl` values at time, so `setvl` sets `vl` to `mvl`. If n is smaller than `mvl`, it should compute only on the last `n` elements in this final iteration of the loop, so `setvl` sets `vl` to `n`. `setvl` also writes another scalar register to help with later loop bookkeeping. Below is the RV64V code for vector DAXPY for any value of `n`.

```
      vsetdcfg  2 DP FP     # Enable 2 64b Fl.Pt. registers
      fld       f0,a        # Load scalar a
loop: setvl     t0,a0       # vl = t0 = min(mvl,n)
      vld       v0,x5       # Load vector X
      slli      t1,t0,3     # t1 = vl * 8 (in bytes)
      add       x5,x5,t1    # Increment pointer to X by vl*8
      vmul      v0,v0,f0    # Vector-scalar mult
      vld       v1,x6       # Load vector Y
      vadd      v1,v0,v1    # Vector-vector add
      sub       a0,a0,t0    # n -= vl (t0)
      vst       v1,x6       # Store the sum into Y
      add       x6,x6,t1    # Increment pointer to Y by vl*8
      bnez      a0,loop     # Repeat if n != 0
      vdisable              # Disable vector regs
```

Predicate Registers: Handling IF Statements in Vector Loops

From Amdahl's law, we know that the speedup on programs with low to moderate levels of vectorization will be very limited. The presence of conditionals (IF statements) inside loops and the use of sparse matrices are two main reasons for lower levels of vectorization. Programs that contain IF statements in loops cannot be run

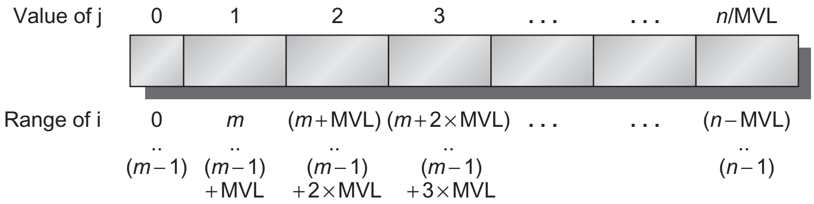


Figure 4.6 A vector of arbitrary length processed with strip mining. All blocks but the first are of length MVL, utilizing the full power of the vector processor. In this figure, we use the variable *m* for the expression (`n % MVL`). (The C operator `%` is modulo.)

in vector mode using the techniques we have discussed up to now because the IF statements introduce control dependences into a loop. Likewise, we cannot implement sparse matrices efficiently using any of the capabilities we have seen so far. We examine strategies for dealing with conditional execution here, leaving the discussion of sparse matrices for later.

Consider the following loop written in C:

```
for (i = 0; i < 64;  i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which X[i] $\neq$ 0, then the subtraction could be vectorized.

The common extension for this capability is *vector-mask control*. In RV64V, predicate registers hold the mask and essentially provide conditional execution of each element operation in a vector instruction. These registers use a Boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a Boolean condition to determine whether to execute a scalar instruction (see Chapter 3). When the predicate register `p0` is set, all following vector instructions operate only on the vector elements whose corresponding entries in the predicate register are 1. The entries in the destination vector register that correspond to a 0 in the mask register are unaffected by the vector operation. Like vector registers, predicate registers are configured and can be disabled. Enabling a predicate register initializes it to all 1 s, meaning that subsequent vector instructions operate on all vector elements. We can now use the following code for the previous loop, assuming that the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively:

```
vsetdcfg    2*FP64     # Enable 2 64b FP vector regs
vsetpcfgi   1          # Enable 1 predicate register
vld         v0,x5      # Load vector X into v0
vld         v1,x6      # Load vector Y into v1
fmv.d.x     f0,x0      # Put (FP) zero into f0
```

```
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst         v0,x5       # Store the result in X
vdisable                # Disable vector registers
vpdisable               # Disable predicate registers
```

Compiler writers use the term *IF-conversion* to transform an IF statement into a straight-line code sequence using conditional execution.

Using a vector-mask register does have overhead, however. With scalar architectures, conditionally executed instructions still require execution time when the condition is not satisfied. Nonetheless, the elimination of a branch and the associated control dependences can make a conditional instruction faster even if it sometimes does useless work. Similarly, vector instructions executed with a vector mask still take the same execution time, even for the elements where the mask is zero. Likewise, despite a significant number of zeros in the mask, using vector-mask control may still be significantly faster than using scalar mode.

As we will see in Section 4.4, one difference between vector processors and GPUs is the way they handle conditional statements. Vector processors make the predicate registers part of the architectural state and rely on compilers to manipulate mask registers explicitly. In contrast, GPUs get the same effect using hardware to manipulate internal mask registers that are invisible to GPU software. In both cases, the hardware spends the time to execute a vector element whether the corresponding mask bit is 0 or 1, so the GFLOPS rate drops when masks are used.

Memory Banks: Supplying Bandwidth for Vector Load/Store Units

The behavior of the load/store vector unit is significantly more complicated than that of the arithmetic functional units. The start-up time for a load is the time to get the first word from memory into a register. If the rest of the vector can be supplied without stalling, then the vector initiation rate is equal to the rate at which new words are fetched or stored. Unlike simpler functional units, the initiation rate may not necessarily be 1 clock cycle because memory bank stalls can reduce effective throughput.

Typically, penalties for start-ups on load/store units are higher than those for arithmetic units—over 100 clock cycles on many processors. For RV64V, we assume a start-up time of 12 clock cycles, the same as the Cray-1. (Recent vector computers use caches to bring down latency of vector loads and stores.)

To maintain an initiation rate of one word fetched or stored per clock cycle, the memory system must be capable of producing or accepting this much data. Spreading accesses across multiple independent memory banks usually delivers the desired rate. As we will soon see, having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

Most vector processors use memory banks, which allow several independent accesses rather than simple memory interleaving for three reasons:

1. Many vector computers support many loads or stores per clock cycle, and the memory bank cycle time is usually several times larger than the processor cycle time. To support simultaneous accesses from multiple loads or stores, the memory system needs multiple banks and needs to be able to control the addresses to the banks independently.
2. Most vector processors support the ability to load or store data words that are not sequential. In such cases, independent bank addressing, rather than interleaving, is required.
3. Most vector computers support multiple processors sharing the same memory system, so each processor will be generating its own separate stream of addresses.

In combination, these features lead to the desire for a large number of independent memory banks, as the following example shows.

Example The largest configuration of a Cray T90 (Cray T932) has 32 processors, each capable of generating 4 loads and 2 stores per clock cycle. The processor clock cycle is 2.167 ns, while the cycle time of the SRAMs used for the memory system is 15 ns. Calculate the minimum number of memory banks required to allow all processors to run at the full memory bandwidth.

Answer The maximum number of memory references each cycle is 192: 32 processors times 6 references per processor. Each SRAM bank is busy for $15/2.167=6.92$ clock cycles, which we round up to 7 processor clock cycles. Therefore we require a minimum of $192\times 7=1344$ memory banks!

The Cray T932 actually has 1024 memory banks, so the early models could not sustain the full bandwidth to all processors simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

Taking a higher-level perspective, vector load/store units play a similar role to prefetch units in scalar processors in that both try to deliver data bandwidth by supplying processors with streams of data.

Stride: Handling Multidimensional Arrays in Vector Architectures

The position in memory of adjacent elements in a vector may not be sequential. Consider this straightforward code for matrix multiply in C:

```
for (i = 0; i < 100;  i=i+1)
  for (j = 0; j < 100;  j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

We could vectorize the multiplication of each row of B with each column of D and strip-mine the inner loop with k as the index variable.

To do so, we must consider how to address adjacent elements in B and adjacent elements in D. When an array is allocated memory, it is linearized and must be laid out in either row-major order (as in C) or column-major order (as in Fortran). This linearization means that either the elements in the row or the elements in the column are not adjacent in memory. For example, the preceding C code allocates in row-major order, so the elements of D that are accessed by iterations in the inner loop are separated by the row size times 8 (the number of bytes per entry) for a total of 800 bytes. In Chapter 2, we saw that blocking could improve locality in cache-based systems. For vector processors without caches, we need another technique to fetch elements of a vector that are not adjacent in memory.

This distance separating elements to be gathered into a single vector register is called the *stride*. In this example, matrix D has a stride of 100 double words (800 bytes), and matrix B would have a stride of 1 double word (8 bytes). For column-major order, which is used by Fortran, the strides would be reversed. Matrix D would have a stride of 1, or 1 double word (8 bytes), separating successive elements, while matrix B would have a stride of 100, or 100 double words (800 bytes). Thus, without reordering the loops, the compiler can't hide the long distances between successive elements for both B and D.

Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements. Thus a vector processor can handle strides greater than one, called *nonunit strides*, using only vector load and vector store operations with stride capability. This ability to access nonsequential memory locations and to reshape them into a dense structure is one of the major advantages of a vector architecture.

Caches inherently deal with unit-stride data; increasing block size can help reduce miss rates for large scientific datasets with unit stride, but increasing block size can even have a negative effect for data that are accessed with nonunit strides. While blocking techniques can solve some of these problems (see Chapter 2), the ability to access noncontiguous data efficiently remains an advantage for vector processors on certain problems, as we will see in Section 4.7.

On RV64V, where the addressable unit is a byte, the stride for our example would be 800. The value must be computed dynamically because the size of the matrix may not be known at compile time or—just like vector length—may change for different executions of the same statement. The vector stride, like the vector starting address, can be put in a general-purpose register. Then the RV64V instruction VLDS (load vector with stride) fetches the vector into a vector register. Likewise, when storing a nonunit stride vector, use the instruction VSTS (store vector with stride).

Supporting strides greater than one complicates the memory system. Once we introduce nonunit strides, it becomes possible to request accesses from the same bank frequently. When multiple accesses contend for a bank, a memory bank conflict occurs, thereby stalling one access. A bank conflict and thus a stall will occur if

$$\frac{\text{Number of banks}}{\text{Least common multiple (Stride, Number of banks)}} < \text{Bank busy time}$$

Example Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

Answer Because the number of banks is larger than the bank busy time, for a stride of 1, the load will take 12 + 64 = 76 clock cycles, or 1.2 clock cycles per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element, slowing it down by a factor of 5!

Gather-Scatter: Handling Sparse Matrices in Vector Architectures

As previously mentioned, sparse matrices are commonplace, so it is important to have techniques to allow programs with sparse matrices to execute in vector mode. In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
for (i = 0; i < n;  i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C. (A and C must have the same number of nonzero elements—`n` of them—so `K` and `M` are the same size.)

The primary mechanism for supporting sparse matrices is *gather-scatter operations* using index vectors. The goal of such operations is to support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., the zeros are included) of a sparse matrix. A *gather* operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector. The result is a dense vector in a vector register. After these elements are operated on in a dense form, the sparse vector can be stored in an expanded form by a *scatter* store, using the same index vector. Hardware support for such operations is called *gather-scatter*, and it appears on nearly all modern vector processors. The RV64V instructions are `vldi` (load vector indexed or gather) and `vsti` (store vector indexed or scatter). For example, if `x5`, `x6`, `x7`, and `x28` contain the starting addresses of the vectors in the previous sequence, we can code the inner loop with vector instructions such as:

```
vsetdcfg  4*FP64        # 4  64b  FP vector registers
vld       v0, x7        # Load K[]
vldx      v1, x5, v0)   # Load A[K[]]
vld       v2, x28       # Load M[]
vldi      v3, x6, v2)   # Load C[M[]]
vadd      v1, v1, v3    # Add them
vstx      v1, x5, v0)   # Store A[K[]]
vdisable                # Disable vector registers
```

This technique allows code with sparse matrices to run in vector mode. A simple vectorizing compiler could not automatically vectorize the preceding source code because the compiler would not know that the elements of K are distinct values, and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it was safe to run the loop in vector mode.

Although indexed loads and stores (gather and scatter) can be pipelined, they typically run much more slowly than nonindexed loads or stores, because the memory banks are not known from the start of the instruction. The register file must also provide communication between the lanes of a vector unit to support gather and scatter.

Each element of a gather or scatter has an individual address, so they can't be handled in groups, and there can be conflicts at many places throughout the memory system. Thus each individual access incurs significant latency even on cache-based systems. However, as Section 4.7 shows, a memory system can deliver better performance by designing for this case and by using more hardware resources versus when architects have a laissez-faire attitude toward such unpredictable accesses.

As we will see in Section 4.4, all loads are gathers and all stores are scatters in GPUs in that no separate instructions restrict addresses to be sequential. To turn the potentially slow gathers and scatters into the more efficient unit-stride accesses to memory, the GPU hardware must recognize the sequential addresses during execution and the GPU programmer to ensure that all the addresses in a gather or scatter are to adjacent locations.

Programming Vector Architectures

An advantage of vector architectures is that compilers can tell programmers at compile time whether a section of code will vectorize or not, often giving hints

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure 4.7 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP (Vajapeyam, 1991). The first column shows the vectorization level obtained with the compiler without hints, and the second column shows the results after the codes have been improved with hints from a team of Cray Research programmers.

as to why it did not vectorize the code. This straightforward execution model allows experts in other domains to learn how to improve performance by revising their code or by giving hints to the compiler when it's okay to assume independence between operations, such as for gather-scatter data transfers. It is this dialogue between the compiler and the programmer, with each side giving hints to the other on how to improve performance, that simplifies programming of vector computers.

Today, the main factor that affects the success with which a program runs in vector mode is the structure of the program itself: Do the loops have true data dependences (see Section 4.5), or can they be restructured so as not to have such dependences? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded.

As an indication of the level of vectorization achievable in scientific programs, let's look at the vectorization levels observed for the Perfect Club benchmarks. Figure 4.7 shows the percentage of operations executed in vector mode for two versions of the code running on the Cray Y-MP. The first version is that obtained with just compiler optimization on the original code, while the second version uses extensive hints from a team of Cray Research programmers. Several studies of the performance of applications on vector processors show a wide variation in the level of compiler vectorization.

The hint-rich versions show significant gains in vectorization level for codes that the compiler could not vectorize well by itself, with all codes now above 50% vectorization. The median vectorization improved from about 70% to about 90%.

4.3 SIMD Instruction Set Extensions for Multimedia

SIMD Multimedia Extensions started with the simple observation that many media applications operate on narrower data types than the 32-bit processors were optimized for. Graphics systems would use 8 bits to represent each of the three primary colors plus 8 bits for transparency. Depending on the application, audio samples are usually represented with 8 or 16 bits. By partitioning the carry chains within, say, a 256-bit adder, a processor could perform simultaneous operations on short vectors of thirty-two 8-bit operands, sixteen 16-bit operands, eight 32-bit operands, or four 64-bit operands. The additional cost of such partitioned adders was small. Figure 4.8 summarizes typical multimedia SIMD instructions. Like vector instructions, a SIMD instruction specifies the same operation on vectors of data. Unlike vector machines with large register files such as the RISC-V RV64V vector registers, which can hold, say, thirty-two 64-bit elements in each of 32 vector registers, SIMD instructions tend to specify fewer operands and thus use much smaller register files.

In contrast to vector architectures, which offer an elegant instruction set that is intended to be the target of a vectorizing compiler, SIMD extensions have three major omissions: no vector length register, no strided or gather/scatter data transfer instructions, and no mask registers.

1. Multimedia SIMD extensions fix the number of data operands in the opcode, which has led to the addition of hundreds of instructions in the MMX, SSE, and AVX extensions of the x86 architecture. Vector architectures have a vector-length register that specifies the number of operands for the current operation. These variable-length vector registers easily accommodate programs that naturally have shorter vectors than the maximum size the architecture supports. Moreover, vector architectures have an implicit maximum vector length in the

Instruction category	**Operands**
Unsigned add/subtract	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Maximum/minimum	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Average	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Shift right/left	Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit
Floating point	Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit

Figure 4.8 Summary of typical SIMD multimedia support for 256-bit-wide operations. Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.

architecture, which combined with the vector length register avoids the use of many opcodes.

2. Up until recently, multimedia SIMD did not offer the more sophisticated addressing modes of vector architectures, namely strided accesses and gather-scatter accesses. These features increase the number of programs that a vector compiler can successfully vectorize (see Section 4.7).
3. Although this is changing, multimedia SIMD usually did not offer the mask registers to support conditional execution of elements as in vector architectures.

Such omissions make it harder for the compiler to generate SIMD code and increase the difficulty of programming in SIMD assembly language.

For the x86 architecture, the MMX instructions added in 1996 repurposed the 64-bit floating-point registers, so the basic instructions could perform eight 8-bit operations or four 16-bit operations simultaneously. These were joined by parallel MAX and MIN operations, a wide variety of masking and conditional instructions, operations typically found in digital signal processors, and ad hoc instructions that were believed to be useful in important media libraries. Note that MMX reused the floating-point data-transfer instructions to access memory.

The Streaming SIMD Extensions (SSE) successor in 1999 added 16 separate registers (`XMM` registers) that were 128 bits wide, so now instructions could simultaneously perform sixteen 8-bit operations, eight 16-bit operations, or four 32-bit operations. It also performed parallel single-precision floating-point arithmetic. Because SSE had separate registers, it needed separate data transfer instructions. Intel soon added double-precision SIMD floating-point data types via SSE2 in 2001, SSE3 in 2004, and SSE4 in 2007. Instructions with four single-precision floating-point operations or two parallel double-precision operations increased the peak floating-point performance of the x86 computers, as long as programmers placed the operands side by side. With each generation, they also added ad hoc instructions whose aim was to accelerate specific multimedia functions perceived to be important.

The Advanced Vector Extensions (AVX), added in 2010, doubled the width of the registers again to 256 bits (`YMM` registers) and thereby offered instructions that double the number of operations on all narrower data types. Figure 4.9 shows AVX instructions useful for double-precision floating-point computations. AVX2 in 2013 added 30 new instructions such as gather (`VGATHER`) and vector shifts (`VPSLL`, `VPSRL`, `VPSRA`). AVX-512 in 2017 doubled the width again to 512 bits (`ZMM` registers), doubled the number of the registers again to 32, and added about 250 new instructions including scatter (`VPSCATTER`) and mask registers (`OPMASK`). AVX includes preparations to extend registers to 1024 bits in future editions of the architecture.

In general, the goal of these extensions has been to accelerate carefully written libraries rather than for the compiler to generate them (see Appendix H), but recent x86 compilers are trying to generate such code, particularly for floating-point-intensive applications. Since the opcode determines the width of the SIMD register, every time the width doubles, so must the number of SIMD instructions.

AVX instruction	Description
`VADDPD`	Add four packed double-precision operands
`VSUBPD`	Subtract four packed double-precision operands
`VMULPD`	Multiply four packed double-precision operands
`VDIVPD`	Divide four packed double-precision operands
`VFMADDPD`	Multiply and add four packed double-precision operands
`VFMSUBPD`	Multiply and subtract four packed double-precision operands
`VCMPxx`	Compare four packed double-precision operands for `EQ`, `NEQ`, `LT`, `LE`, `GT`, `GE`, ...
`VMOVAPD`	Move aligned four packed double-precision operands
`VBROADCASTSD`	Broadcast one double-precision operand to four locations in a 256-bit register

Figure 4.9 AVX instructions for x86 architecture useful in double-precision floating-point programs. Packed-double for 256-bit AVX means four 64-bit operands executed in SIMD mode. As the width increases with AVX, it is increasingly important to add data permutation instructions that allow combinations of narrow operands from different parts of the wide registers. AVX includes instructions that shuffle 32-bit, 64-bit, or 128-bit operands within a 256-bit register. For example, BROADCAST replicates a 64-bit operand four times in an AVX register. AVX also includes a large variety of fused multiply-add/subtract instructions; we show just two here.

Given these weaknesses, why are multimedia SIMD extensions so popular? First, they initially cost little to add to the standard arithmetic unit and they were easy to implement. Second, they require scant extra processor state compared to vector architectures, which is always a concern for context switch times. Third, you need a lot of memory bandwidth to support a vector architecture, which many computers don't have. Fourth, SIMD does not have to deal with problems in virtual memory when a single instruction can generate 32 memory accesses and any of which can cause a page fault. The original SIMD extensions used separate data transfers per SIMD group of operands that are aligned in memory, and so they cannot cross page boundaries. Another advantage of short, fixed-length "vectors" of SIMD is that it is easy to introduce instructions that can help with new media standards, such as instructions that perform permutations or instructions that consume either fewer or more operands than vectors can produce. Finally, there was concern about how well vector architectures can work with caches. More recent vector architectures have addressed all of these problems. The overarching issue, however, is that due the overiding importance of backwards binary compatability, once an architecture gets started on the SIMD path it's very hard to get off it.

Example To get an idea about what multimedia instructions look like, assume we added a 256-bit SIMD multimedia instruction extension to RISC-V, tentatively called RVP for "packed." We concentrate on floating-point in this example. We add the suffix "`4D`" on instructions that operate on four double-precision operands at once. Like vector architectures, you can think of a SIMD Processor as having lanes, four in this case. RV64P expands the F registers to be the full width, in this case 256 bits. This example shows the RISC-V SIMD code for the `DAXPY` loop, with the changes to the RISC-V code for SIMD underlined. We assume that the starting addresses of `X` and `Y` are in `x5` and `x6`, respectively.

Answer Here is the RISC-V SIMD code:

```
      fld      f0,a         #Load scalar a
      splat.4D f0,f0        #Make 4 copies of a
      addi     x28,x5,#256  #Last address to load
Loop: fld.4D   f1,0(x5)     #Load X[i] ... X[i+3]
      fmul.4D  f1,f1,f0     #a×X[i] ... a×X[i+3]
      fld.4D   f2,0(x6)     #Load Y[i] ... Y[i+3]
      fadd.4D  f2,f2,f1     # a×X[i]+Y[i]...
                            # a×X[i+3]+Y[i+3]
      fsd.4D   f2,0(x6)     #Store Y[i]... Y[i+3]
      addi     x5,x5,#32    #Increment index to X
      addi     x6,x6,#32    #Increment index to Y
      bne      x28,x5,Loop  #Check if done
```

The changes were replacing every RISC-V double-precision instruction with its `4D` equivalent, increasing the increment from 8 to 32, and adding the `splat` instruction that makes 4 copies of `a` in the 256 bits of `f0`. While not as dramatic as the 32 × reduction of dynamic instruction bandwidth of RV64V, RISC-V SIMD does get almost a 4 × reduction: 67 versus 258 instructions executed for RV64G. This code knows the number of elements. That number is often determined at run time, which would require an extra strip-mine loop to handle the case when the number is not a modulo of 4.

Programming Multimedia SIMD Architectures

Given the ad hoc nature of the SIMD multimedia extensions, the easiest way to use these instructions has been through libraries or by writing in assembly language.

Recent extensions have become more regular, giving compilers a more reasonable target. By borrowing techniques from vectorizing compilers, compilers are starting to produce SIMD instructions automatically. For example, advanced compilers today can generate SIMD floating-point instructions to deliver much higher performance for scientific codes. However, programmers must be sure to align all the data in memory to the width of the SIMD unit on which the code is run to prevent the compiler from generating scalar instructions for otherwise vectorizable code.

The Roofline Visual Performance Model

One visual, intuitive way to compare potential floating-point performance of variations of SIMD architectures is the Roofline model (Williams et al., 2009). The horizontal and diagonal lines of the graphs it produces give this simple model its name and indicate its value (see Figure 4.11). It ties together floating-point performance, memory performance, and arithmetic intensity in a two-dimensional graph.

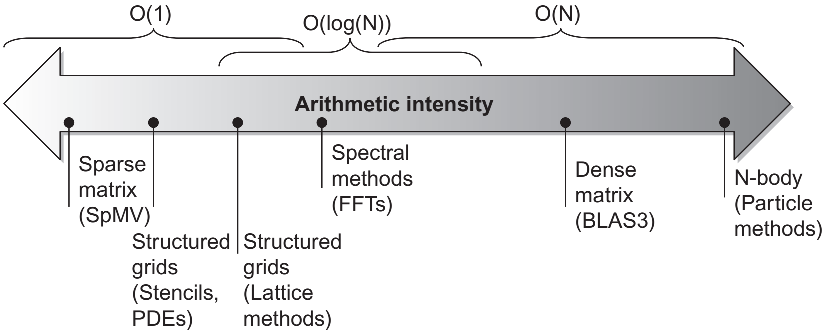


Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory (Williams et al., 2009). Some kernels have an arithmetic intensity that scales with problem size, such as a dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

Arithmetic intensity is the ratio of floating-point operations per byte of memory accessed. It can be calculated by taking the total number of floating-point operations for a program divided by the total number of data bytes transferred to main memory during program execution. Figure 4.10 shows the relative arithmetic intensity of several example kernels.

Peak floating-point performance can be found using the hardware specifications. Many of the kernels in this case study do not fit in on-chip caches, so peak memory performance is defined by the memory system behind the caches. Note that we need the peak memory bandwidth that is available to the processors, not just at the DRAM pins as in Figure 4.27 on page 328. One way to find the (delivered) peak memory performance is to run the Stream benchmark.

Figure 4.11 shows the Roofline model for the NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer on the right. The vertical Y-axis is achievable floating-point performance from 2 to 256 GFLOPS/s. The horizontal X-axis is arithmetic intensity, varying from 1/8 FLOP/DRAM byte accessed to 16 FLOP/DRAM byte accessed in both graphs. Note that the graph is a log-log scale, and that Rooflines are done just once for a computer.

For a given kernel, we can find a point on the X-axis based on its arithmetic intensity. If we drew a vertical line through that point, the performance of the kernel on that computer must lie somewhere along that line. We can plot a horizontal line showing peak floating-point performance of the computer. Obviously, the actual floating-point performance can be no higher than the horizontal line because that is a hardware limit.

How could we plot the peak memory performance? Because the X-axis is FLOP/byte and the Y-axis is FLOP/s, bytes/s is just a diagonal line at a 45-degree angle in this figure. Thus we can plot a third line that gives the maximum floating-point performance that the memory system of that computer can support for a given

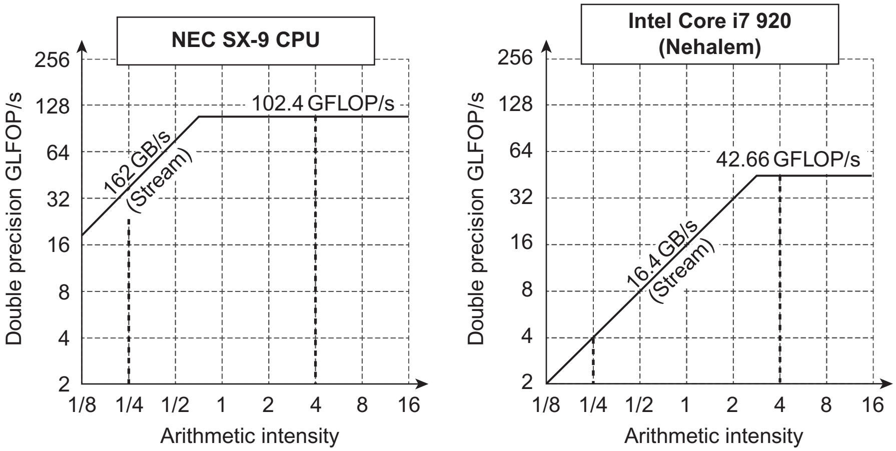


Figure 4.11 Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD extensions on the right (Williams et al., 2009). This Roofline is for unit-stride memory accesses and double-precision floating-point performance. NEC SX-9 is a vector supercomputer announced in 2008 that cost millions of dollars. It has a peak DP FP performance of 102.4 GFLOP/s and a peak memory bandwidth of 162 GB/s from the Stream benchmark. The Core i7 920 has a peak DP FP performance of 42.66 GFLOP/s and a peak memory bandwidth of 16.4 GB/s. The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte show that both processors operate at peak performance. In this case, the SX-9 at 102.4 FLOP/s is 2.4 × faster than the Core i7 at 42.66 GFLOP/s. At an arithmetic intensity of 0.25 FLOP/byte, the SX-9 is 10 × faster at 40.5 GFLOP/s versus 4.1 GFLOP/s for the Core i7.

arithmetic intensity. We can express the limits as a formula to plot these lines in the graphs in Figure 4.11:

$$\begin{aligned}\text{Attainable GFLOPs/s} = \text{Min}(&\text{Peak Memory BW}\\ &\times \text{Arithmetic Intensity, Peak Floating} - \text{Point Perf.})\end{aligned}$$

The "Roofline" sets an upper bound on performance of a kernel depending on its arithmetic intensity. If we think of arithmetic intensity as a pole that hits the roof, either it hits the flat part of the roof, which means performance is computationally limited, or it hits the slanted part of the roof, which means performance is ultimately limited by memory bandwidth. In Figure 4.11, the vertical dashed line on the right (arithmetic intensity of 4) is an example of the former and the vertical dashed line on the left (arithmetic intensity of 1/4) is an example of the latter. Given a Roofline model of a computer, you can apply it repeatedly, because it doesn't vary by kernel.

Note that the "ridge point," where the diagonal and horizontal roofs meet, offers an interesting insight into a computer. If it is far to the right, then only kernels with very high arithmetic intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance. As we will see, this vector processor has both much higher memory bandwidth and a ridge point far to the left as compared to other SIMD Processors.

Figure 4.11 shows that the peak computational performance of the SX-9 is 2.4 × faster than Core i7, but the memory performance is 10 × faster. For programs with an arithmetic intensity of 0.25, the SX-9 is 10 × faster (40.5 versus 4.1 GFLOP/s). The higher memory bandwidth moves the ridge point from 2.6 in the Core i7 to 0.6 on the SX-9, which means many more programs can reach the peak computational performance on the vector processor.

4.4 Graphics Processing Units

People can buy a GPU chip with thousands of parallel floating-point units for a few hundred dollars and plug it into their desk side PC. Such affordability and convenience makes high performance computing available to many. The interest in GPU computing blossomed when this potential was combined with a programming language that made GPUs easier to program. Therefore many programmers of scientific and multimedia applications today are pondering whether to use GPUs or CPUs. For programmers interested in machine learning, which is the subject of Chapter 7, GPUs are currently the preferred platform.

GPUs and CPUs do not go back in computer architecture genealogy to a common ancestor; there is no "missing link" that explains both. As Section 4.10 describes, the primary ancestors of GPUs are graphics accelerators, as doing graphics well is the reason why GPUs exist. While GPUs are moving toward mainstream computing, they can't abandon their responsibility to continue to excel at graphics. Thus the design of GPUs may make more sense when architects ask, given the hardware invested to do graphics well, how can we supplement it to improve the performance of a wider range of applications?

Note that this section concentrates on using GPUs for computing. To see how GPU computing combines with the traditional role of graphics acceleration, see "Graphics and Computing GPUs," by John Nickolls and David Kirk (Appendix A in the 5th edition of *Computer Organization and Design* by the same authors as this book).

Because the terminology and some hardware features are quite different from vector and SIMD architectures, we believe it will be easier if we start with the simplified programming model for GPUs before we describe the architecture.

Programming the GPU

CUDA is an elegant solution to the problem of representing parallelism in algorithms, not all algorithms, but enough to matter. It seems to resonate in some way with the way we think and code, allowing an easier, more natural expression of parallelism beyond the task level.

Vincent Natol,
"Kudos for CUDA," *HPC Wire* (2010)

The challenge for the GPU programmer is not simply getting good performance on the GPU, but also in coordinating the scheduling of computation on the system processor and the GPU and the transfer of data between system memory and GPU memory. Moreover, as we see will see later in this section, GPUs have virtually every type of parallelism that can be captured by the programming environment: multithreading, MIMD, SIMD, and even instruction-level.

NVIDIA decided to develop a C-like language and programming environment that would improve the productivity of GPU programmers by attacking both the challenges of heterogeneous computing and of multifaceted parallelism. The name of their system is *CUDA*, for Compute Unified Device Architecture. CUDA produces C/C++ for the system processor (*host*) and a C and C++ dialect for the GPU (*device*, thus the D in CUDA). A similar programming language is *OpenCL*, which several companies are developing to offer a vendor-independent language for multiple platforms.

NVIDIA decided that the unifying theme of all these forms of parallelism is the *CUDA Thread*. Using this lowest level of parallelism as the programming primitive, the compiler and the hardware can gang thousands of CUDA Threads together to utilize the various styles of parallelism within a GPU: multithreading, MIMD, SIMD, and instruction-level parallelism. Therefore NVIDIA classifies the CUDA programming model as single instruction, multiple thread (*SIMT*). For reasons we will soon see, these threads are blocked together and executed in groups of threads, called a *Thread Block*. We call the hardware that executes a whole block of threads a *multithreaded SIMD Processor*.

We need just a few details before we can give an example of a CUDA program:

- To distinguish between functions for the GPU (device) and functions for the system processor (host), CUDA uses `__device__` or `__global__` for the former and `__host__` for the latter.
- CUDA variables declared with `__device__` are allocated to the GPU Memory (see below), which is accessible by all multithreaded SIMD Processors.
- The extended function call syntax for the function *name* that runs on the GPU is

  name < `<<dimGrid, dimBlock>>` > (… *parameter list*…)

  where `dimGrid` and `dimBlock` specify the dimensions of the code (in Thread Blocks) and the dimensions of a block (in threads).
- In addition to the identifier for blocks (`blockIdx`) and the identifier for each thread in a block (`threadIdx`), CUDA provides a keyword for the number of threads per block (`blockDim`), which comes from the `dimBlock` parameter in the preceding bullet.

Before seeing the CUDA code, let's start with conventional C code for the `DAXPY` loop from Section 4.2:

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
          for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

Following is the CUDA version. We launch `n` threads, one per vector element, with 256 CUDA Threads per Thread Block in a multithreaded SIMD Processor. The GPU function starts by calculating the corresponding element index `i` based on the block ID, the number of threads per block, and the thread ID. As long as this index is within the array (`i < n`), it performs the multiply and add.

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

Comparing the C and CUDA codes, we see a common pattern to parallelizing data-parallel CUDA code. The C version has a loop where each iteration is independent from the others, allowing the loop to be transformed straightforwardly into a parallel code where each loop iteration becomes a separate thread. (As previously mentioned and described in detail in Section 4.5, vectorizing compilers also rely on a lack of dependences between iterations of a loop, which are called *loop-carried dependences*.) The programmer determines the parallelism in CUDA explicitly by specifying the grid dimensions and the number of threads per SIMD Processor. By assigning a single thread to each element, there is no need to synchronize between threads when writing results to memory.

The GPU hardware handles parallel execution and thread management; it is not done by applications or by the operating system. To simplify scheduling by the hardware, CUDA requires that Thread Blocks be able to execute independently and in any order. Different Thread Blocks cannot communicate directly, although they can *coordinate* using atomic memory operations in global memory.

As we will soon see, many GPU hardware concepts are not obvious in CUDA. Writing efficient GPU code requires that programmers think in terms of SIMD operations, even though the CUDA programming model looks like MIMD. Performance programmers must keep the GPU hardware in mind when writing in CUDA. That could hurt programmer productivity, but then most programmers are using GPUs instead of CPUs to get performance. For reasons explained shortly, they know that they need to keep groups of 32 threads together in control flow to get the best performance from multithreaded SIMD Processors and to create many more threads per multithreaded SIMD Processor to hide latency to DRAM. They also need to keep the data addresses localized in one or a few blocks of memory to get the expected memory performance.

Like many parallel systems, a compromise between productivity and performance is for CUDA to include intrinsics to give programmers explicit control over the hardware. The struggle between productivity on the one hand versus allowing the programmer to be able to express anything that the hardware can do on the other hand happens often in parallel computing. It will be interesting to see how the language evolves in this classic productivity-performance battle as well as to see whether CUDA becomes popular for other GPUs or even other architectural styles.

NVIDIA GPU Computational Structures

The uncommon heritage mentioned above helps explain why GPUs have their own architectural style and their own terminology independent from CPUs. One obstacle to understanding GPUs has been the jargon, with some terms even having misleading names. This obstacle has been surprisingly difficult to overcome, as the many rewrites of this chapter can attest.

To try to bridge the twin goals of making the architecture of GPUs understandable *and* learning the many GPU terms with nontraditional definitions, our approach is to use the CUDA terminology for software but initially use more descriptive terms for the hardware, sometimes borrowing terms from OpenCL. Once we explain the GPU architecture in our terms, we'll map them into the official jargon of NVIDIA GPUs.

From left to right, Figure 4.12 lists the descriptive term used in this section, the closest term from mainstream computing, the official NVIDIA GPU jargon in case you are interested, and then a short explanation of the term. The rest of this section explains the microarchitectural features of GPUs using the descriptive terms on the left in the figure.

We use NVIDIA systems as our example as they are representative of GPU architectures. Specifically, we follow the terminology of the preceding CUDA parallel programming language and use the NVIDIA Pascal GPU as the example (see Section 4.7).

Like vector architectures, GPUs work well only with data-level parallel problems. Both styles have gather-scatter data transfers and mask registers, and GPU processors have even more registers than do vector processors. Sometimes, GPUs implement certain features in hardware that vector processors would implement in software. This difference is because vector processors have a scalar processor that can execute a software function. Unlike most vector architectures,

Type	Descriptive name	Closest old term outside of GPUs	Official CUDA/NVIDIA GPU term	Short explanation
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via local memory
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it only contains SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors
	SIMD Thread Scheduler	Thread Scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full Thread Block (body of vectorized loop)

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: program abstractions, machine objects, processing hardware, and memory hardware. Figure 4.21 on page 312 associates vector terms with the closest terms here, and Figure 4.24 on page 317 and Figure 4.25 on page 318 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

GPUs also rely on multithreading within a single multithreaded SIMD Processor to hide memory latency (see Chapters 2 and 3). However, efficient code for both vector architectures and GPUs requires programmers to think in groups of SIMD operations.

A *Grid* is the code that runs on a GPU that consists of a set of *Thread Blocks*. Figure 4.12 draws the analogy between a grid and a vectorized loop and between a Thread Block and the body of that loop (after it has been strip-mined, so that it is a full computation loop). To give a concrete example, let's suppose we want to multiply two vectors together, each 8192 elements long: `A = B * C`. We'll return to this example throughout this section. Figure 4.13 shows the relationship between this example and these first two GPU terms. The GPU code that works on the whole 8192 element multiply is called a *Grid* (or vectorized loop). To break it down into more manageable sizes, a Grid is composed of *Thread Blocks* (or body of a vectorized loop), each with up to 512 elements. Note that a SIMD instruction executes 32 elements at a time. With 8192 elements in the vectors, this example thus has 16 Thread Blocks because $16 = 8192 \div 512$. The Grid and Thread Block are programming abstractions implemented in GPU hardware that help programmers organize their CUDA code. (The Thread Block is analogous to a strip-mined vector loop with a vector length of 32.)

A Thread Block is assigned to a processor that executes that code, which we call a *multithreaded SIMD Processor*, by the *Thread Block Scheduler*. The programmer tells the Thread Block Scheduler, which is implemented in hardware, how many Thread Blocks to run. In this example, it would send 16 Thread Blocks to multithreaded SIMD Processors to compute all 8192 elements of this loop

Figure 4.14 shows a simplified block diagram of a multithreaded SIMD Processor. It is similar to a vector processor, but it has many parallel functional units instead of a few that are deeply pipelined, as in a vector processor. In the programming example in Figure 4.13, each multithreaded SIMD Processor is assigned 512 elements of the vectors to work on. SIMD Processors are full processors with separate PCs and are programmed using threads (see Chapter 3).

The GPU hardware then contains a collection of multithreaded SIMD Processors that execute a Grid of Thread Blocks (bodies of vectorized loop); that is, a GPU is a multiprocessor composed of multithreaded SIMD Processors.

A GPU can have from one to several dozen multithreaded SIMD Processors. For example, the Pascal P100 system has 56, while the smaller chips may have as few as one or two. To provide transparent scalability across models of GPUs with a differing number of multithreaded SIMD Processors, the Thread Block Scheduler assigns Thread Blocks (bodies of a vectorized loop) to multithreaded SIMD Processors. Figure 4.15 shows the floor plan of the P100 implementation of the Pascal architecture.

Dropping down one more level of detail, the machine object that the hardware creates, manages, schedules, and executes is a *thread of SIMD instructions*. It is a traditional thread that contains exclusively SIMD instructions. These threads of SIMD instructions have their own PCs, and they run on a multithreaded SIMD Processor. The *SIMD Thread Scheduler* knows which threads of SIMD instructions are ready to run and then sends them off to a dispatch unit to be run on

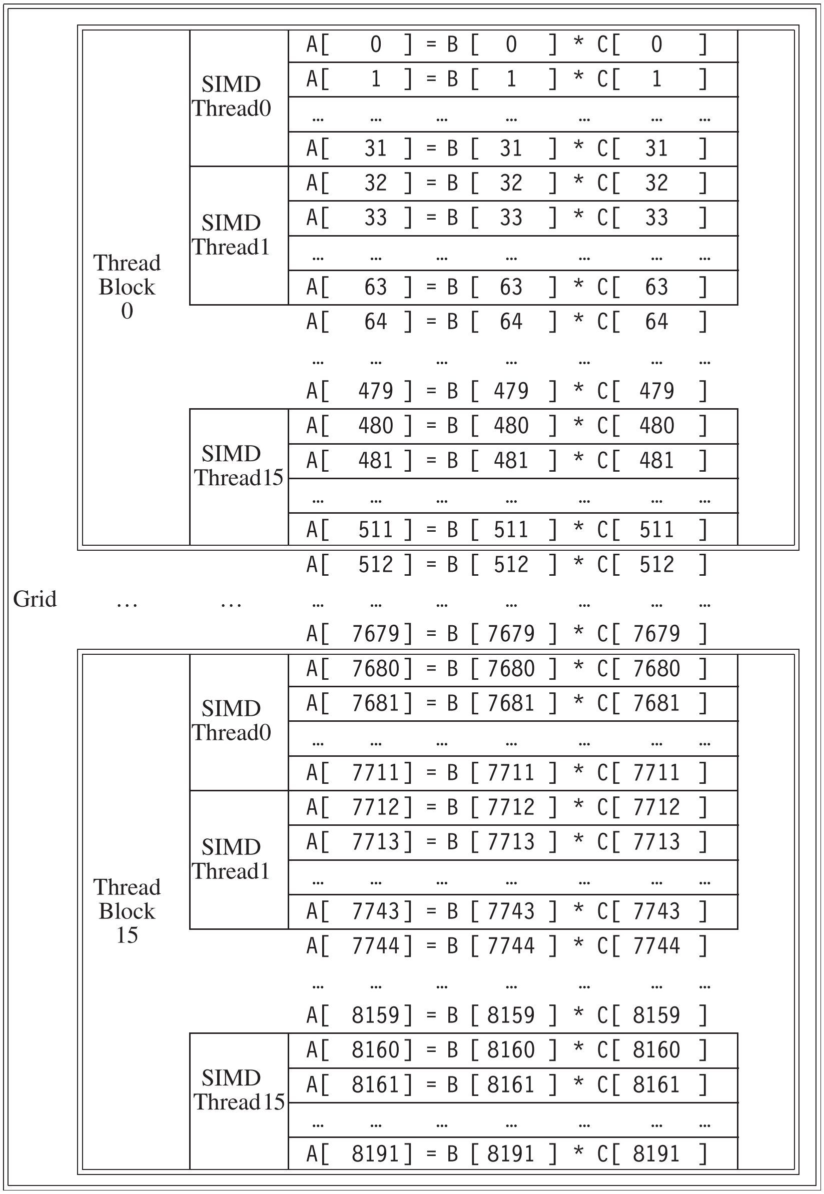

Grid
Thread Block 0
SIMD Thread0
A[0] = B [0] * C[0]
A[1] = B [1] * C[1]
… … … … … … …
A[31] = B [31] * C[31]
SIMD Thread1
A[32] = B [32] * C[32]
A[33] = B [33] * C[33]
… … … … … … …
A[63] = B [63] * C[63]
A[64] = B [64] * C[64]
… … … … … … …
A[479] = B [479] * C[479]
SIMD Thread15
A[480] = B [480] * C[480]
A[481] = B [481] * C[481]
… … … … … … …
A[511] = B [511] * C[511]
A[512] = B [512] * C[512]
… …
… … … … … … …
A[7679] = B [7679] * C[7679]
Thread Block 15
SIMD Thread0
A[7680] = B [7680] * C[7680]
A[7681] = B [7681] * C[7681]
… … … … … … …
A[7711] = B [7711] * C[7711]
SIMD Thread1
A[7712] = B [7712] * C[7712]
A[7713] = B [7713] * C[7713]
… … … … … … …
A[7743] = B [7743] * C[7743]
A[7744] = B [7744] * C[7744]
… … … … … … …
A[8159] = B [8159] * C[8159]
SIMD Thread15
A[8160] = B [8160] * C[8160]
A[8161] = B [8161] * C[8161]
… … … … … … …
A[8191] = B [8191] * C[8191]

Figure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIMD instructions to a vector-vector multiply, with each vector being 8192 elements long. Each thread of SIMD instructions calculates 32 elements per instruction, and in this example, each Thread Block contains 16 threads of SIMD instructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks to multithreaded SIMD Processors, and the hardware Thread Scheduler picks which thread of SIMD instructions to run each clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via local memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 32 for Pascal GPUs.)

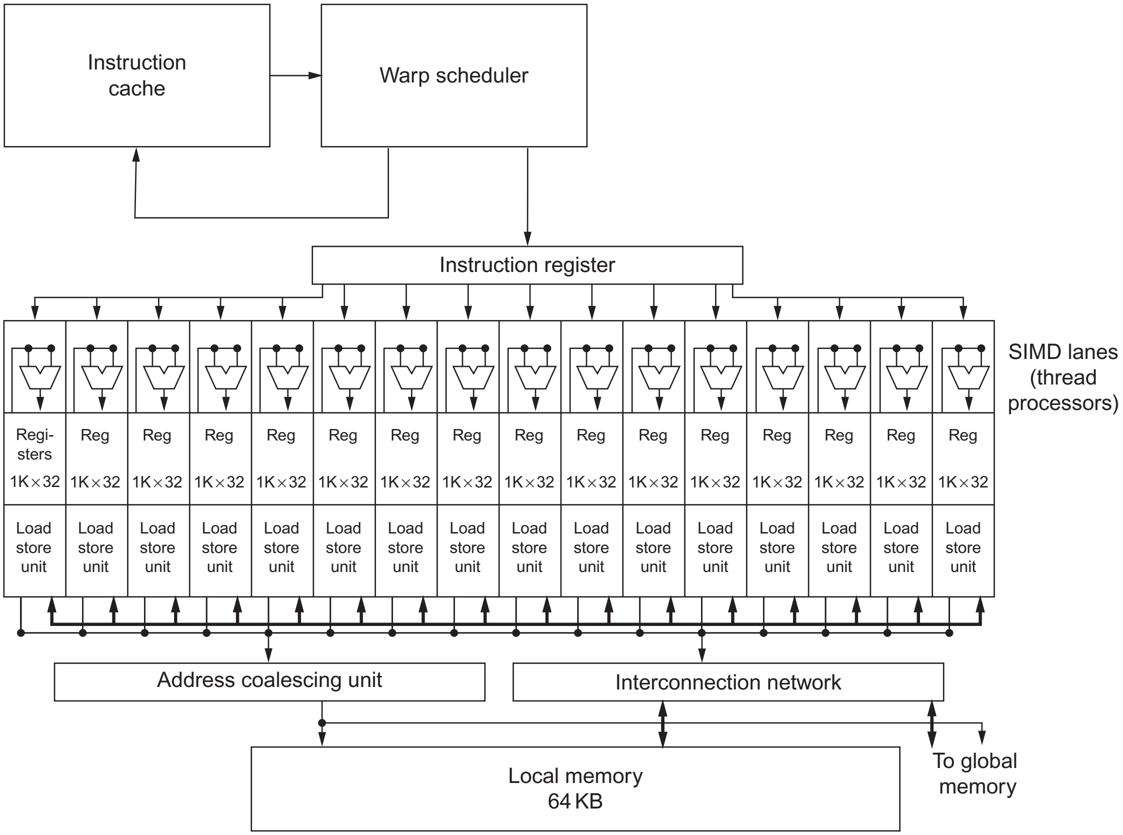

Instruction cache
Warp scheduler
Instruction register
SIMD lanes (thread processors)
Registers
Reg
1K×32
Load store unit
Address coalescing unit
Interconnection network
Local memory 64 KB
To global memory

Figure 4.14 Simplified block diagram of a multithreaded SIMD Processor. It has 16 SIMD Lanes. The SIMD Thread Scheduler has, say, 64 independent threads of SIMD instructions that it schedules with a table of 64 program counters (PCs). Note that each lane has 1024 32-bit registers.

the multithreaded SIMD Processor. Thus GPU hardware has two levels of hardware schedulers: (1) the *Thread Block Scheduler* that assigns Thread Blocks (bodies of vectorized loops) to multithreaded SIMD Processors and (2) the SIMD Thread Scheduler *within* a SIMD Processor, which schedules when threads of SIMD instructions should run.

The SIMD instructions of these threads are 32 wide, so each thread of SIMD instructions in this example would compute 32 of the elements of the computation. In this example, Thread Blocks would contain 512/32 = 16 SIMD Threads (see Figure 4.13).

Because the thread consists of SIMD instructions, the SIMD Processor must have parallel functional units to perform the operation. We call them *SIMD Lanes*, and they are quite similar to the Vector Lanes in Section 4.2.

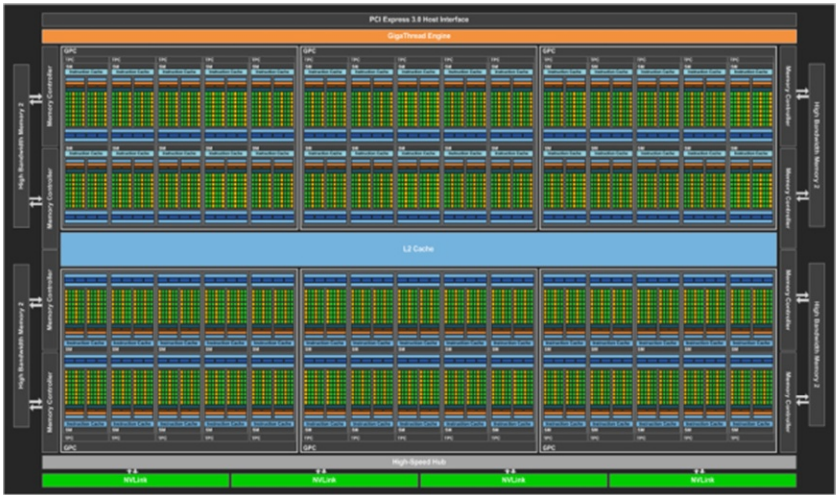


Figure 4.15 Full-chip block diagram of the Pascal P100 GPU. It has 56 multithreaded SIMD Processors, each with an L1 cache and local memory, 32 L2 units, and a memory-bus width of 4096 data wires. (It has 60 blocks, with four spares to improve yield.) The P100 has 4 HBM2 ports supporting up to 16 GB of capacity. It contains 15.4 billion transistors.

With the Pascal GPU, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes 2 clock cycles to complete. Each thread of SIMD instructions is executed in lock step and scheduled only at the beginning. Staying with the analogy of a SIMD Processor as a vector processor, you could say that it has 16 lanes, the vector length is 32, and the chime is 2 clock cycles. (This wide but shallow nature is why we use the more accurate term SIMD Processor rather than vector.)

Note that the number of lanes in a GPU SIMD Processor can be anything up to the number of threads in a Thread Block, just as the number of lanes in a vector processor can vary between 1 and the maximum vector length. For example, across GPU generations, the number of lanes per SIMD Processor has fluctuated between 8 and 32.

Because by definition the threads of SIMD instructions are independent, the SIMD Thread Scheduler can pick whatever thread of SIMD instructions is ready, and need not stick with the next SIMD instruction in the sequence within a thread. The SIMD Thread Scheduler includes a scoreboard (see Chapter 3) to keep track of up to 64 threads of SIMD instructions to see which SIMD instruction is ready to go. The latency of memory instructions is variable because of hits and misses in the caches and the TLB, thus the requirement of a scoreboard to determine when these instructions are complete. Figure 4.16 shows the SIMD Thread Scheduler picking threads of SIMD instructions in a different order over time. The assumption of GPU architects is that GPU applications have so many threads of SIMD instructions that multithreading can both hide the latency to DRAM and increase utilization of multithreaded SIMD Processors.

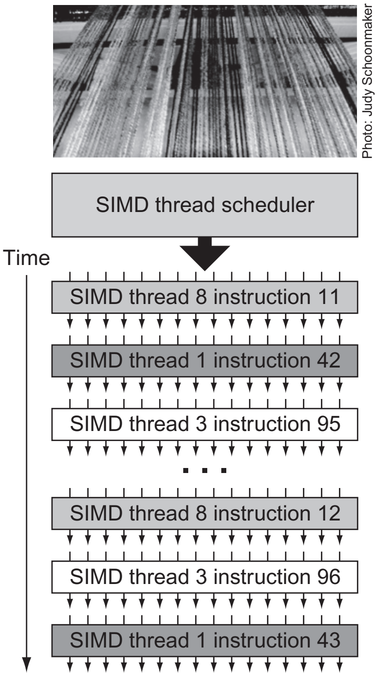


Photo: Judy Schoonmaker
SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43

Figure 4.16 Scheduling of threads of SIMD instructions. The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD Thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD Thread each time.

Continuing our vector multiply example, each multithreaded SIMD Processor must load 32 elements of two vectors from memory into registers, perform the multiply by reading and writing registers, and store the product back from registers into memory. To hold these memory elements, a SIMD Processor has between an impressive 32,768–65,536 32-bit registers (1024 per lane in Figure 4.14), depending on the model of the Pascal GPU. Just like a vector processor, these registers are divided logically across the Vector Lanes or, in this case, SIMD Lanes.

Each SIMD Thread is limited to no more than 256 registers, so you might think of a SIMD Thread as having up to 256 vector registers, with each vector register having 32 elements and each element being 32 bits wide. (Because double-precision floating-point operands use two adjacent 32-bit registers, an alternative view is that each SIMD Thread has 128 vector registers of 32 elements, each of which is 64 bits wide.)

There is a trade-off between register use and maximum number of threads; fewer registers per thread means more threads are possible, and more registers mean fewer threads. That is, not all SIMD Threads need to have the maximum number of registers. Pascal architects believe much of this precious silicon area would be idle if all threads had the maximum number of registers.

To be able to execute many threads of SIMD instructions, each is dynamically allocated a set of the physical registers on each SIMD Processor when threads of SIMD instructions are created and freed when the SIMD Thread exits. For example, a programmer can have a Thread Block that uses 36 registers per thread with, say, 16 SIMD Threads alongside another Thread Block that has 20 registers per thread with 32 SIMD Threads. Subsequent Thread Blocks may show up in any order, and the registers have to be allocated on demand. While this variability can lead to fragmentation and make some registers unavailable, in practice most Thread Blocks use the same number of registers for a given vectorizable loop ("grid"). The hardware must know where the registers for each Thread Block are in the large register file, and this is recorded on a per Thread-Block basis. This flexibility requires routing, arbitration, and banking in the hardware because a specific register for a given Thread Block could end up in any location in the register file.

Note that a CUDA Thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by one SIMD Lane. Beware that CUDA Threads are very different from POSIX Threads; you can't make arbitrary system calls from a CUDA Thread.

We're now ready to see what GPU instructions look like.

NVIDA GPU Instruction Set Architecture

Unlike most system processors, the instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set. *PTX* (*Parallel Thread Execution*) provides a stable instruction set for compilers as well as compatibility across generations of GPUs. The hardware instruction set is hidden from the programmer. PTX instructions describe the operations on a single CUDA Thread and usually map one-to-one with hardware instructions, but one PTX instruction can expand to many machine instructions, and vice versa. PTX uses an unlimited number of write-once registers and the compiler must run a register allocation procedure to map the PTX registers to a fixed number of read-write hardware registers available on the actual device. The optimizer runs subsequently and can reduce register use even further. This optimizer also eliminates dead code, folds instructions together, and calculates places where branches might diverge and places where diverged paths could converge.

Although there is some similarity between the x86 microarchitecture and PTX, in that both translate to an internal form (microinstructions for x86), the difference is that this translation happens in hardware at runtime during execution on the x86 versus in software and load time on a GPU.

The format of a PTX instruction is

```
opcode.type d, a, b, c;
```

where `d` is the destination operand; `a`, `b`, and `c` are source operands; and the operation type is one of the following:

Type	.type specifier
Untyped bits 8, 16, 32, and 64 bits	`.b8`, `.b16`, `.b32`, `.b64`
Unsigned integer 8, 16, 32, and 64 bits	`.u8`, `.u16`, `.u32`, `.u64`
Signed integer 8, 16, 32, and 64 bits	`.s8`, `.s16`, `.s32`, `.s64`
Floating Point 16, 32, and 64 bits	`.f16`, `.f32`, `.f64`

Source operands are 32-bit or 64-bit registers or a constant value. Destinations are registers, except for store instructions.

Figure 4.17 shows the basic PTX instruction set. All instructions can be predicated by 1-bit predicate registers, which can be set by a set predicate instruction (`setp`). The control flow instructions are functions `call` and `return`, thread `exit`, `branch`, and barrier synchronization for threads within a Thread Block `(bar.sync)`. Placing a predicate in front of a branch instruction gives us conditional branches. The compiler or PTX programmer declares virtual registers as 32-bit or 64-bit typed or untyped values. For example, `R0`, `R1`, `...` are for 32-bit values and `RD0`, `RD1`, `...` are for 64-bit registers. Recall that the assignment of virtual registers to physical registers occurs at load time with PTX.

The following sequence of PTX instructions is for one iteration of our DAXPY loop on page 292:

```
shl.u32 R8, blockIdx, 8   ; Thread Block ID * Block size
                          ;(256 or 2^8)
add.u32 R8, R8, threadIdx ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3         ; byte offset
ld.global.f64 RD0, [X+R8]; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]; RD2 = Y[i]
mul.f64 RD0, RD0, RD4     ; Product in RD0 = RD0 * RD4
                          ; (scalar a)
add.f64 RD0, RD0, RD2     ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0; Y[i] = sum (X[i]*a + Y[i])
```

As demonstrated above, the CUDA programming model assigns one CUDA Thread to each loop iteration and offers a unique identifier number to each Thread Block (`blockIdx`) and one to each CUDA Thread within a block (`threadIdx`). Thus it creates 8192 CUDA Threads and uses the unique number to address each element within the array, so there is no incrementing or branching code. The first three PTX instructions calculate that unique element byte offset in `R8`, which is added to the base of the arrays. The following PTX instructions load two double-precision floating-point operands, multiply and add them, and store the sum. (We'll describe the PTX code corresponding to the CUDA code "`if (i < n)`" below.)

Note that unlike vector architectures, GPUs don't have separate instructions for sequential data transfers, strided data transfers, and gather-scatter data transfers.

Group	Instruction	Example	Meaning	Comments
Arithmetic	arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64			
	add.type	add.f32 d, a, b	d = a + b;	
	sub.type	sub.f32 d, a, b	d = a – b;	
	mul.type	mul.f32 d, a, b	d = a * b;	
	mad.type	mad.f32 d, a, b, c	d = a * b + c;	multiply-add
	div.type	div.f32 d, a, b	d = a / b;	multiple microinstructions
	rem.type	rem.u32 d, a, b	d = a % b;	integer remainder
	abs.type	abs.f32 d, a	d = |a|;	
	neg.type	neg.f32 d, a	d = 0 – a;	
	min.type	min.f32 d, a, b	d = (a < b)? a:b;	floating selects non-NaN
	max.type	max.f32 d, a, b	d = (a > b)? a:b;	floating selects non-NaN
	setp.cmp.type	setp.lt.f32 p, a, b	p = (a < b);	compare and set predicate
	numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan			
	mov.type	mov.b32 d, a	d = a;	move
	selp.type	selp.f32 d, a, b, p	d = p? a: b;	select with predicate
	cvt.dtype.atype	cvt.f32.s32 d, a	d = convert(a);	convert atype to dtype
Special function	special .type = .f32 (some .f64)			
	rcp.type	rcp.f32 d, a	d = 1/a;	reciprocal
	sqrt.type	sqrt.f32 d, a	d = sqrt(a);	square root
	rsqrt.type	rsqrt.f32 d, a	d = 1/sqrt(a);	reciprocal square root
	sin.type	sin.f32 d, a	d = sin(a);	sine
	cos.type	cos.f32 d, a	d = cos(a);	cosine
	lg2.type	lg2.f32 d, a	d = log(a)/log(2)	binary logarithm
	ex2.type	ex2.f32 d, a	d = 2 ** a;	binary exponential
Logical	logic.type = .pred,.b32, .b64			
	and.type	and.b32 d, a, b	d = a & b;	
	or.type	or.b32 d, a, b	d = a | b;	
	xor.type	xor.b32 d, a, b	d = a ^b;	
	not.type	not.b32 d, a, b	d = ~a;	one's complement
	cnot.type	cnot.b32 d, a, b	d = (a==0)? 1:0;	C logical not
	shl.type	shl.b32 d, a, b	d = a << b;	shift left
	shr.type	shr.s32 d, a, b	d = a >> b;	shift right
Memory access	memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64			
	ld.space.type	ld.global.b32 d, [a+off]	d = *(a+off);	load from memory space
	st.space.type	st.shared.b32 [d+off], a	*(d+off) = a;	store to memory space
	tex.nd.dtyp.btype	tex.2d.v4.f32.f32 d, a, b	d = tex2d(a, b);	texture lookup
	atom.spc.op.type	atom.global.add.u32 d,[a], b atom.global.cas.b32 d,[a], b, c	atomic { d = *a; *a = op(*a, b); }	atomic read-modify-write operation
	atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32			
Control flow	branch	@p bra target	if (p) goto target;	conditional branch
	call	call (ret), func, (params)	ret = func(params);	call function
	ret	ret	return;	return from function call
	bar.sync	bar.sync d	wait for threads	barrier synchronization
	exit	exit	exit;	terminate thread execution

Figure 4.17 Basic PTX GPU thread instructions.

All data transfers are gather-scatter! To regain the efficiency of sequential (unit-stride) data transfers, GPUs include special Address Coalescing hardware to recognize when the SIMD Lanes within a thread of SIMD instructions are collectively issuing sequential addresses. That runtime hardware then notifies the Memory Interface Unit to request a block transfer of 32 sequential words. To get this important performance improvement, the GPU programmer must ensure that adjacent CUDA Threads access nearby addresses at the same time so that they can be coalesced into one or a few memory or cache blocks, which our example does.

Conditional Branching in GPUs

Just like the case with unit-stride data transfers, there are strong similarities between how vector architectures and GPUs handle IF statements, with the former implementing the mechanism largely in software with limited hardware support and the latter making use of even more hardware. As we will see, in addition to explicit predicate registers, GPU branch hardware uses internal masks, a branch synchronization stack, and instruction markers to manage when a branch diverges into multiple execution paths and when the paths converge.

At the PTX assembler level, control flow of one CUDA Thread is described by the PTX instructions branch, call, return, and exit, plus individual per-thread-lane predication of each instruction, specified by the programmer with per-thread-lane 1-bit predicate registers. The PTX assembler analyzes the PTX branch graph and optimizes it to the fastest GPU hardware instruction sequence. Each can make its own decision on a branch and does not need to be in lock step.

At the GPU hardware instruction level, control flow includes branch, jump, jump indexed, call, call indexed, return, exit, and special instructions that manage the branch synchronization stack. GPU hardware provides each SIMD Thread with its own stack; a stack entry contains an identifier token, a target instruction address, and a target thread-active mask. There are GPU special instructions that push stack entries for a SIMD Thread and special instructions and instruction markers that pop a stack entry or unwind the stack to a specified entry and branch to the target instruction address with the target thread-active mask. GPU hardware instructions also have an individual per-lane predication (enable/disable), specified with a 1-bit predicate register for each lane.

The PTX assembler typically optimizes a simple outer-level IF-THEN-ELSE statement coded with PTX branch instructions to solely predicated GPU instructions, without any GPU branch instructions. A more complex control flow often results in a mixture of predication and GPU branch instructions with special instructions and markers that use the branch synchronization stack to push a stack entry when some lanes branch to the target address, while others fall through. NVIDIA says a branch *diverges* when this happens. This mixture is also used when a SIMD Lane executes a synchronization marker or *converges*, which pops a stack entry and branches to the stack-entry address with the stack-entry thread-active mask.

The PTX assembler identifies loop branches and generates GPU branch instructions that branch to the top of the loop, along with special stack instructions to handle individual lanes breaking out of the loop and converging the SIMD Lanes when all lanes have completed the loop. GPU indexed jump and indexed call instructions push entries on the stack so that when all lanes complete the switch statement or function call, the SIMD Thread converges.

A GPU set predicate instruction (`setp` in Figure 4.17) evaluates the conditional part of the IF statement. The PTX branch instruction then depends on that predicate. If the PTX assembler generates predicated instructions with no GPU branch instructions, it uses a per-lane predicate register to enable or disable each SIMD Lane for each instruction. The SIMD instructions in the threads inside the THEN part of the IF statement broadcast operations to all the SIMD Lanes. Those lanes with the predicate set to 1 perform the operation and store the result, and the other SIMD Lanes don't perform an operation or store a result. For the ELSE statement, the instructions use the complement of the predicate (relative to the THEN statement), so the SIMD Lanes that were idle now perform the operation and store the result while their formerly active siblings don't. At the end of the ELSE statement, the instructions are unpredicated so the original computation can proceed. Thus, for equal length paths, an IF-THEN-ELSE operates at 50% efficiency or less.

IF statements can be nested, thus the use of a stack, and the PTX assembler typically generates a mix of predicated instructions and GPU branch and special synchronization instructions for complex control flow. Note that deep nesting can mean that most SIMD Lanes are idle during execution of nested conditional statements. Thus, doubly nested IF statements with equal-length paths run at 25% efficiency, triply nested at 12.5% efficiency, and so on. The analogous case would be a vector processor operating where only a few of the mask bits are ones.

Dropping down a level of detail, the PTX assembler sets a "branch synchronization" marker on appropriate conditional branch instructions that pushes the current active mask on a stack inside each SIMD Thread. If the conditional branch diverges (some lanes take the branch but some fall through), it pushes a stack entry and sets the current internal active mask based on the condition. A branch synchronization marker pops the diverged branch entry and flips the mask bits before the ELSE portion. At the end of the IF statement, the PTX assembler adds another branch synchronization marker that pops the prior active mask off the stack into the current active mask.

If all the mask bits are set to 1, then the branch instruction at the end of the THEN skips over the instructions in the ELSE part. There is a similar optimization for the THEN part in case all the mask bits are 0 because the conditional branch jumps over the THEN instructions. Parallel IF statements and PTX branches often use branch conditions that are unanimous (all lanes agree to follow the same path) such that the SIMD Thread does not diverge into a different individual lane control flow. The PTX assembler optimizes such branches to skip over blocks of instructions that are not executed by any lane of a SIMD Thread. This optimization is useful in conditional error checking, for example, where the test must be made but is rarely taken.

The code for a conditional statement similar to the one in Section 4.2 is

```
if (X[i] != 0)
  X[i] = X[i] - Y[i];
else X[i] = Z[i];
```

This IF statement could compile to the following PTX instructions (assuming that `R8` already has the scaled thread ID), with *`*Push`*, *`*Comp`*, *`*Pop`* indicating the branch synchronization markers inserted by the PTX assembler that push the old mask, complement the current mask, and pop to restore the old mask:

```
   ld.global.f64 RD0, [X+R8]       ; RD0 = X[i]
   setp.neq.s32 P1, RD0, #0        ;P1 is predicate reg 1
   @!P1, bra ELSE1, *Push          ; Push old mask, set new
                         ; mask bits if P1 false, go to ELSE1
   ld.global.f64 RD2, [Y+R8]       ; RD2 = Y[i]
   sub.f64 RD0, RD0, RD2           ; Difference in RD0
   st.global.f64 [X+R8], RD0       ; X[i] = RD0
   @P1, bra ENDIF1, *Comp          ; complement mask bits
                                   ; if P1 true, go to ENDIF1
 ELSE1: ld.global.f64 RD0, [Z+R8]  ; RD0 = Z[i]
        st.global.f64 [X+R8], RD0  ; X[i] = RD0
ENDIF1:<next instruction>, *Pop    ; pop to restore old mask
```

Once again, normally all instructions in the IF-THEN-ELSE statement are executed by a SIMD Processor. It's just that only some of the SIMD Lanes are enabled for the THEN instructions and some lanes for the ELSE instructions. As previously mentioned, in the surprisingly common case that the individual lanes agree on the predicated branch—such as branching on a parameter value that is the same for all lanes so that all active mask bits are 0s or all are 1s—the branch skips the THEN instructions or the ELSE instructions.

This flexibility makes it appear that an element has its own program counter; however, in the slowest case, only one SIMD Lane could store its result every 2 clock cycles, with the rest idle. The analogous slowest case for vector architectures is operating with only one mask bit set to 1. This flexibility can lead naive GPU programmers to poor performance, but it can be helpful in the early stages of program development. Keep in mind, however, that the only choice for a SIMD Lane in a clock cycle is to perform the operation specified in the PTX instruction or be idle; two SIMD Lanes cannot simultaneously execute different instructions.

This flexibility also helps explain the name *CUDA Thread* given to each element in a thread of SIMD instructions, because it gives the illusion of acting independently. A naive programmer may think that this thread abstraction means GPUs handle conditional branches more gracefully. Some threads go one way, the rest go another, which seems true as long as you're not in a hurry. Each CUDA Thread is either executing the same instruction as every other thread in the Thread Block or it is idle. This synchronization makes it easier to handle loops with conditional branches because the mask capability can turn off SIMD Lanes and it detects the end of the loop automatically.

The resulting performance sometimes belies that simple abstraction. Writing programs that operate SIMD Lanes in this highly independent MIMD mode is like writing programs that use lots of virtual address space on a computer with a smaller physical memory. Both are correct, but they may run so slowly that the programmer will not be pleased with the result.

Conditional execution is a case where GPUs do in runtime hardware what vector architectures do at compile time. Vector compilers do a double IF-conversion, generating four different masks. The execution is basically the same as GPUs, but there are some more overhead instructions executed for vectors. Vector architectures have the advantage of being integrated with a scalar processor, allowing them to avoid the time for the 0 cases when they dominate a calculation. Although it will depend on the speed of the scalar processor versus the vector processor, the crossover point when it's better to use scalar might be when less than 20% of the mask bits are 1s. One optimization available at runtime for GPUs, but not at compile time for vector architectures, is to skip the THEN or ELSE parts when mask bits are all 0s or all 1s.

Thus the efficiency with which GPUs execute conditional statements comes down to how frequently the branches will diverge. For example, one calculation of eigenvalues has deep conditional nesting, but measurements of the code show that around 82% of clock cycle issues have between 29 and 32 out of the 32 mask bits set to 1, so GPUs execute this code more efficiently than one might expect.

Note that the same mechanism handles the strip-mining of vector loops—when the number of elements doesn't perfectly match the hardware. The example at the beginning of this section shows that an IF statement checks to see if this SIMD Lane element number (stored in `R8` in the preceding example) is less than the limit `(i < n)`, and it sets masks appropriately.

NVIDIA GPU Memory Structures

Figure 4.18 shows the memory structures of an NVIDIA GPU. Each SIMD Lane in a multithreaded SIMD Processor is given a private section of off-chip DRAM, which we call the *private memory*. It is used for the stack frame, for spilling registers, and for private variables that don't fit in the registers. SIMD Lanes do *not* share private memories. GPUs cache this private memory in the L1 and L2 caches to aid register spilling and to speed up function calls.

We call the on-chip memory that is local to each multithreaded SIMD Processor *local memory*. It is a small scratchpad memory with low latency (a few dozen clocks) and high bandwidth (128 bytes/clock) where the programmer can store data that needs to be reused, either by the same thread or another thread in the same

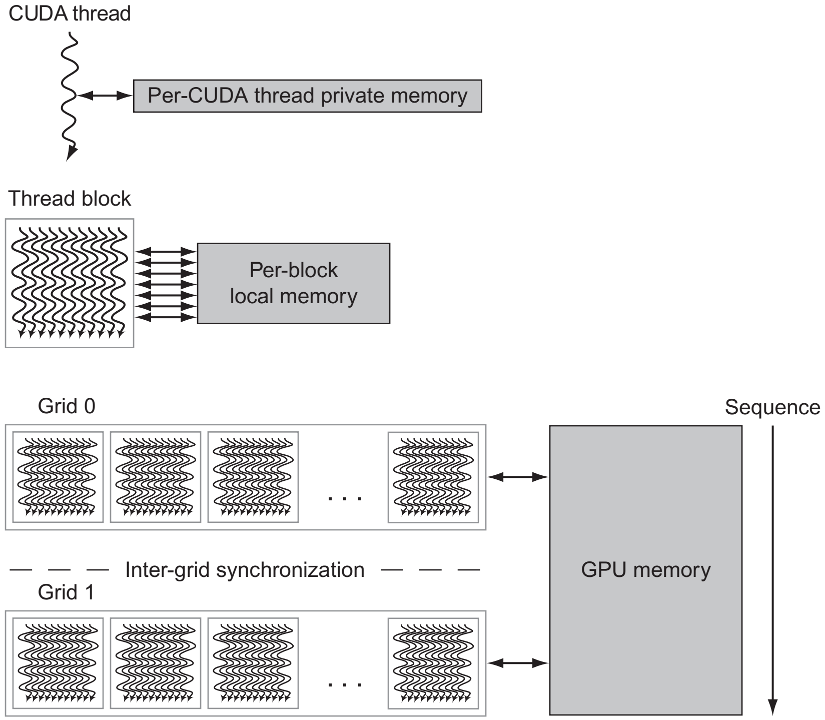


Figure 4.18 GPU memory structures. GPU memory is shared by all Grids (vectorized loops), local memory is shared by all threads of SIMD instructions within a Thread Block (body of a vectorized loop), and private memory is private to a single CUDA Thread. Pascal allows preemption of a Grid, which requires that all local and private memory be able to be saved in and restored from global memory. For completeness sake, the GPU can also access CPU memory via the PCIe bus. This path is commonly used for a final result when its address is in host memory. This option eliminates a final copy from the GPU memory to the host memory.

Thread Block. Local memory is limited in size, typically to 48 KiB. It carries no state between Thread Blocks executed on the same processor. It is shared by the SIMD Lanes within a multithreaded SIMD Processor, but this memory is not shared between multithreaded SIMD Processors. The multithreaded SIMD Processor dynamically allocates portions of the local memory to a Thread Block when it creates the Thread Block, and frees the memory when all the threads of the Thread Block exit. That portion of local memory is private to that Thread Block.

Finally, we call the off-chip DRAM shared by the whole GPU and all Thread Blocks *GPU Memory*. Our vector multiply example used only GPU Memory.

The system processor, called the *host*, can read or write GPU Memory. Local memory is unavailable to the host, as it is private to each multithreaded SIMD Processor. Private memories are unavailable to the host as well.

Rather than rely on large caches to contain the whole working sets of an application, GPUs traditionally use smaller streaming caches and, because their working sets can be hundreds of megabytes, rely on extensive multithreading of threads of SIMD instructions to hide the long latency to DRAM. Given the use of multithreading to hide DRAM latency, the chip area used for large L2 and L3 caches in system processors is spent instead on computing resources and on the large number of registers to hold the state of many threads of SIMD instructions. In contrast, as mentioned, vector loads and stores amortize the latency across many elements because they pay the latency only once and then pipeline the rest of the accesses.

Although hiding memory latency behind many threads was the original philosophy of GPUs and vector processors, all recent GPUs and vector processors have caches to reduce latency. The argument follows Little's Law from queuing theory: the longer the latency, the more threads need to run during a memory access, which in turn requires more registers. Thus GPU caches are added to lower average latency and thereby mask potential shortages of the number of registers.

To improve memory bandwidth and reduce overhead, as mentioned, PTX data transfer instructions in cooperation with the memory controller coalesce individual parallel thread requests from the same SIMD Thread together into a single memory block request when the addresses fall in the same block. These restrictions are placed on the GPU program, somewhat analogous to the guidelines for system processor programs to engage hardware prefetching (see Chapter 2). The GPU memory controller will also hold requests and send ones together to the same open page to improve memory bandwidth (see Section 4.6). Chapter 2 describes DRAM in sufficient detail for readers to understand the potential benefits of grouping related addresses.

Innovations in the Pascal GPU Architecture

The multithreaded SIMD Processor of Pascal is more complicated than the simplified version in Figure 4.20. To increase hardware utilization, each SIMD Processor has two SIMD Thread Schedulers, each with multiple instruction dispatch units (some GPUs have four thread schedulers). The dual SIMD Thread Scheduler selects two threads of SIMD instructions and issues one instruction from each to two sets of 16 SIMD Lanes, 16 load/store units, or 8 special function units. With multiple execution units available, two threads of SIMD instructions are scheduled each clock cycle, allowing 64 lanes to be active. Because the threads are independent, there is no need to check for data dependences in the instruction stream. This innovation would be analogous to a multithreaded vector processor that can issue vector instructions from two independent threads. Figure 4.19 shows the Dual Scheduler issuing instructions, and Figure 4.20 shows the block diagram of the multithreaded SIMD Processor of a Pascal GP100 GPU.

Each new generation of GPU typically adds some new features that increase performance or make it easier for programmers. Here are the four main innovations of Pascal:

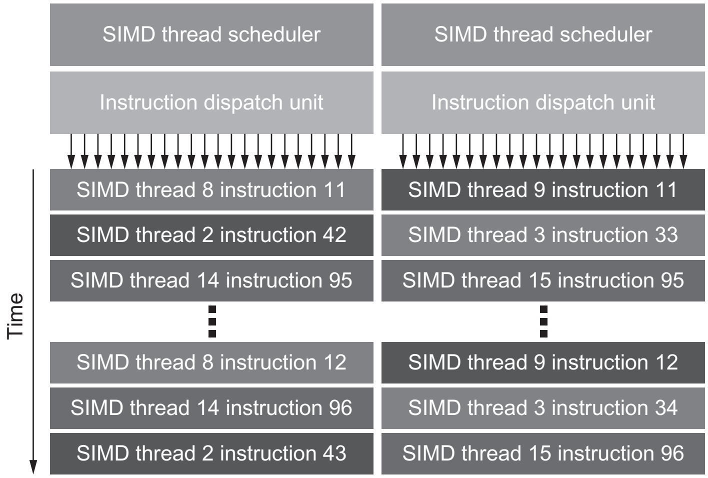

SIMD thread scheduler
SIMD thread scheduler
Instruction dispatch unit
Instruction dispatch unit
Time
SIMD thread 8 instruction 11
SIMD thread 9 instruction 11
SIMD thread 2 instruction 42
SIMD thread 3 instruction 33
SIMD thread 14 instruction 95
SIMD thread 15 instruction 95
SIMD thread 8 instruction 12
SIMD thread 9 instruction 12
SIMD thread 14 instruction 96
SIMD thread 3 instruction 34
SIMD thread 2 instruction 43
SIMD thread 15 instruction 96

Figure 4.19 Block diagram of Pascal's dual SIMD Thread scheduler. Compare this design to the single SIMD Thread design in Figure 4.16.

- *Fast single-precision, double-precision, and half-precision floating-point arithmetic*—Pascal GP100 chip has significant floating-point performance in three sizes, all part of the IEEE standard for floating-point. The single-precision floating-point of the GPU runs at a peak of 10 TeraFLOP/s. Double-precision is roughly half-speed at 5 TeraFLOP/s, and half-precision is about double-speed at 20 TeraFLOP/s when expressed as 2-element vectors. The atomic memory operations include floating-point add for all three sizes. Pascal GP100 is the first GPU with such high performance for half-precision.
- *High-bandwidth memory*—The next innovation of the Pascal GP100 GPU is the use of stacked, high-bandwidth memory (*HBM2*). This memory has a wide bus with 4096 data wires running at 0.7 GHz offering a peak bandwidth of 732 GB/s, which is more than twice as fast as previous GPUs.
- *High-speed chip-to-chip interconnect*—Given the coprocessor nature of GPUs, the PCI bus can be a communications bottleneck when trying to use multiple GPUs with one CPU. Pascal GP100 introduces the *NVLink* communications channel that supports data transfers of up to 20 GB/s in each direction. Each GP100 has 4 NVLink channels, providing a peak aggregate chip-to-chip bandwidth of 160 GB/s per chip. Systems with 2, 4, and 8 GPUs are available for multi-GPU applications, where each GPU can perform load, store, and atomic operations to any GPU connected by NVLink. Additionally, an NVLink channel can communicate with the CPU in some cases. For example, the IBM Power9 CPU supports CPU-GPU communication. In this chip, NVLink provides a coherent view of memory between all GPUs and CPUs connected together. It also provides cache-to-cache communication instead of memory-to-memory communication.

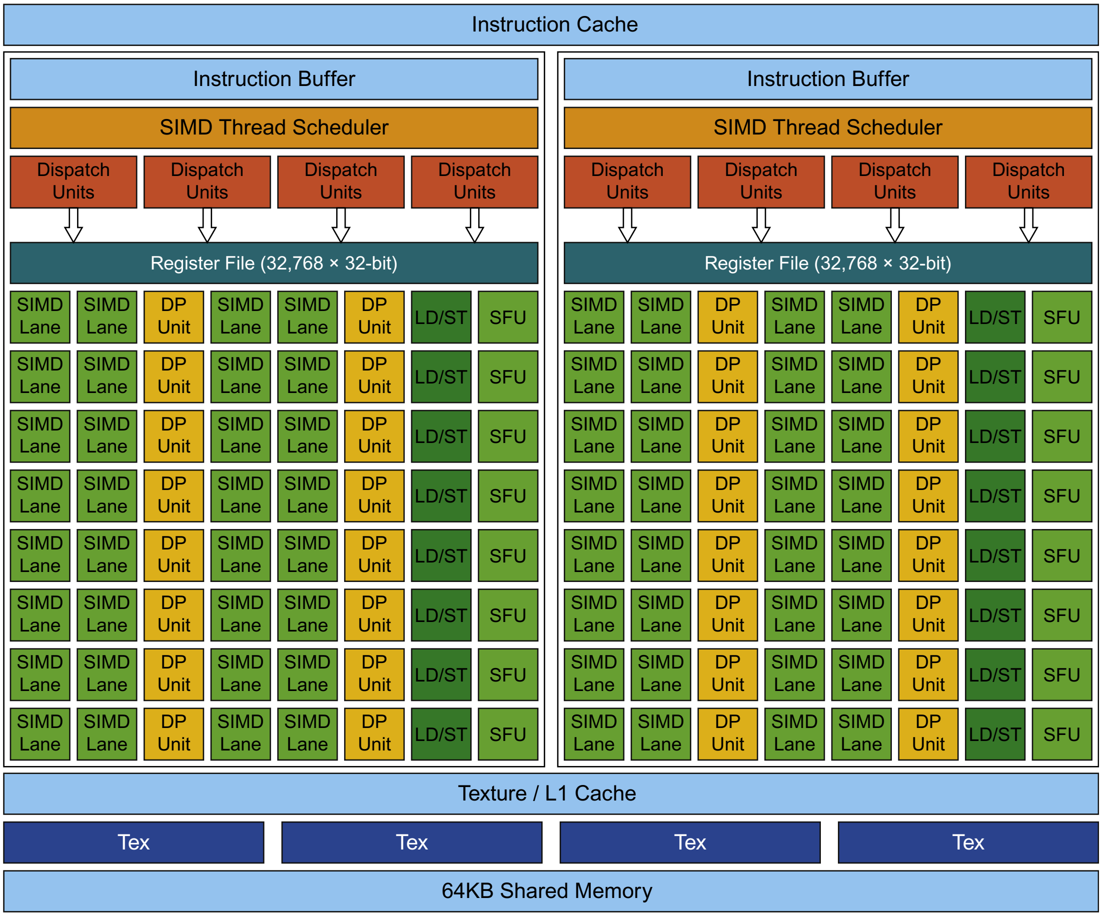

Instruction Cache
Instruction Buffer
SIMD Thread Scheduler
Dispatch Units
Register File (32,768 × 32-bit)
SIMD Lane
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory

Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Pascal GPU. Each of the 64 SIMD Lanes (cores) has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The 64 SIMD Lanes interact with 32 double-precision ALUs (DP units) that perform 64-bit floating-point arithmetic, 16 load-store units (LD/STs), and 16 special function units (SFUs) that calculate functions such as square roots, reciprocals, sines, and cosines.

- *Unified virtual memory and paging support*—The Pascal GP100 GPU adds page-fault capabilities within a unified virtual address space. This feature allows a single virtual address for every data structure that is identical across all the GPUs and CPUs in a single system. When a thread accesses an address that is remote, a page of memory is transferred to the local GPU for subsequent use. Unified memory simplifies the programming model by providing demand paging instead of explicit memory copying between the CPU and GPU or between GPUs. It also allows allocating far more memory than exists on the GPU to solve problems with large memory requirements. As with any virtual memory system, care must be taken to avoid excessive page movement.

Similarities and Differences Between Vector Architectures and GPUs

As we have seen, there really are many similarities between vector architectures and GPUs. Along with the quirky jargon of GPUs, these similarities have contributed to the confusion in architecture circles about how novel GPUs really are. Now that you've seen what is under the covers of vector computers and GPUs, you can appreciate both the similarities and the differences. Because both architectures are designed to execute data-level parallel programs, but take different paths, this comparison is in depth in order to provide a better understanding of what is needed for DLP hardware. Figure 4.21 shows the vector term first and then the closest equivalent in a GPU.

A SIMD Processor is like a vector processor. The multiple SIMD Processors in GPUs act as independent MIMD cores, just as many vector computers have multiple vector processors. This view will consider the NVIDIA Tesla P100 as a 56-core machine with hardware support for multithreading, where each core has 64 lanes. The biggest difference is multithreading, which is fundamental to GPUs and missing from most vector processors.

Looking at the registers in the two architectures, the RV64V register file in our implementation holds entire vectors—that is, a contiguous block of elements. In contrast, a single vector in a GPU will be distributed across the registers of all SIMD Lanes. A RV64V processor has 32 vector registers with perhaps 32 elements, or 1024 elements total. A GPU thread of SIMD instructions has up to 256 registers with 32 elements each, or 8192 elements. These extra GPU registers support multithreading.

Figure 4.22 is a block diagram of the execution units of a vector processor on the left and a multithreaded SIMD Processor of a GPU on the right. For pedagogic purposes, we assume the vector processor has four lanes and the multithreaded SIMD Processor also has four SIMD Lanes. This figure shows that the four SIMD Lanes act in concert much like a four-lane vector unit, and that a SIMD Processor acts much like a vector processor.

In reality, there are many more lanes in GPUs, so GPU "chimes" are shorter. While a vector processor might have 2 to 8 lanes and a vector length of, say, 32—making a chime 4 to 16 clock cycles—a multithreaded SIMD Processor might have 8 or 16 lanes. A SIMD Thread is 32 elements wide, so a GPU chime would just be 2 or 4 clock cycles. This difference is why we use "SIMD Processor" as the more descriptive term because it is closer to a SIMD design than it is to a traditional vector processor design.

Type	Vector term	Closest CUDA/NVIDIA GPU term	Comment
Program abstractions	Vectorized Loop	Grid	Concepts are similar, with the GPU using the less descriptive term
	Chime	—	Because a vector instruction (PTX instruction) takes just 2 cycles on Pascal to complete, a chime is short in GPUs. Pascal has two execution units that support the most common floating-point instructions that are used alternately, so the effective issue rate is 1 instruction every clock cycle
Machine objects	Vector Instruction	PTX Instruction	A PTX instruction of a SIMD Thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction
	Gather/Scatter	Global load/store (ld.global/st.global)	All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it
	Mask Registers	Predicate Registers and Internal Mask Registers	Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically
Processing and memory hardware	Vector Processor	Multithreaded SIMD Processor	These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not
	Control Processor	Thread Block Scheduler	The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide in vector architectures
	Scalar Processor	System Processor	Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture
	Vector Lane	SIMD Lane	Very similar; both are essentially functional units with registers
	Vector Registers	SIMD Lane Registers	The equivalent of a vector register is the same register in all 16 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD Thread is flexible, but the maximum is 256 in Pascal, so the maximum number of vector registers is 256
	Main Memory	GPU Memory	Memory for GPU versus system memory in vector case

Figure 4.21 GPU equivalent to vector terms.

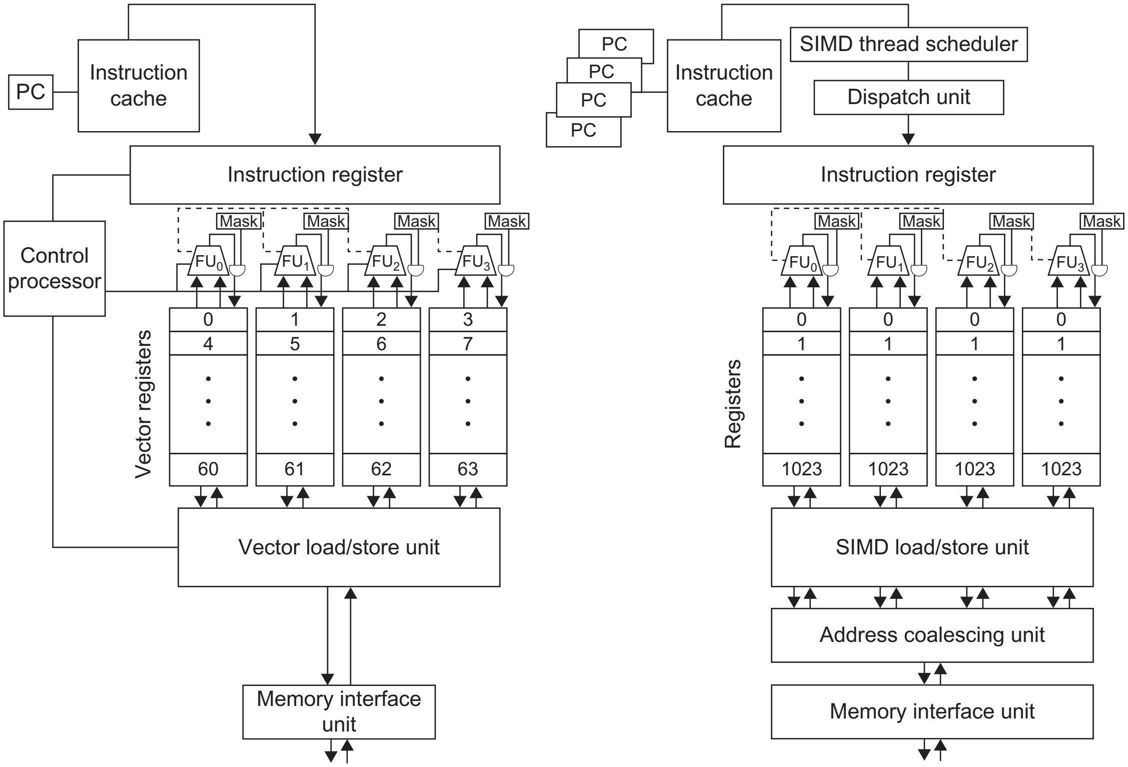

PC
Instruction cache
Instruction register
Control processor
Mask
FU_0
FU_1
FU_2
FU_3
Vector registers
0
1
2
3
4
5
6
7
60
61
62
63
Vector load/store unit
Memory interface unit
PC
PC
PC
PC
Instruction cache
SIMD thread scheduler
Dispatch unit
Instruction register
Registers
1023
SIMD load/store unit
Address coalescing unit
Memory interface unit

Figure 4.22 A vector processor with four lanes on the left and a multithreaded SIMD Processor of a GPU with four SIMD Lanes on the right. (GPUs typically have 16 or 32 SIMD Lanes.) The Control Processor supplies scalar operands for scalar-vector operations, increments addressing for unit and nonunit stride accesses to memory, and performs other accounting-type operations. Peak memory performance occurs only in a GPU when the Address Coalescing Unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD Thread to help with multithreading.

The closest GPU term to a vectorized loop is Grid, and a PTX instruction is the closest to a vector instruction because a SIMD Thread broadcasts a PTX instruction to all SIMD Lanes.

With respect to memory access instructions in the two architectures, all GPU loads are gather instructions and all GPU stores are scatter instructions. If data addresses of CUDA Threads refer to nearby addresses that fall into the same cache/memory block at the same time, the Address Coalescing Unit of the GPU will ensure high memory bandwidth. The *explicit* unit-stride load and store instructions of vector architectures versus the *implicit* unit stride of GPU programming is why writing efficient GPU code requires that programmers think in terms of SIMD operations, even though the CUDA programming model looks like MIMD. Because CUDA Threads can generate their own addresses, strided as well as gather-scatter, addressing vectors are found in both vector architectures and GPUs.

As we mentioned several times, the two architectures take very different approaches to hiding memory latency. Vector architectures amortize it across all the elements of the vector by having a deeply pipelined access, so you pay the latency only once per vector load or store. Therefore vector loads and stores are like a block transfer between memory and the vector registers. In contrast, GPUs hide memory latency using multithreading. (Some researchers are investigating adding multithreading to vector architectures to try to capture the best of both worlds.)

With respect to conditional branch instructions, both architectures implement them using mask registers. Both conditional branch paths occupy time and/or space even when they do not store a result. The difference is that the vector compiler manages mask registers explicitly in software while the GPU hardware and assembler manages them implicitly using branch synchronization markers and an internal stack to save, complement, and restore masks.

The Control Processor of a vector computer plays an important role in the execution of vector instructions. It broadcasts operations to all the Vector Lanes and broadcasts a scalar register value for vector-scalar operations. It also does implicit calculations that are explicit in GPUs, such as automatically incrementing memory addresses for unit-stride and nonunit-stride loads and stores. The Control Processor is missing in the GPU. The closest analogy is the Thread Block Scheduler, which assigns Thread Blocks (bodies of vector loop) to multithreaded SIMD Processors. The runtime hardware mechanisms in a GPU that both generate addresses and then discover if they are adjacent, which is commonplace in many DLP applications, are likely less power-efficient than using a Control Processor.

The scalar processor in a vector computer executes the scalar instructions of a vector program; that is, it performs operations that would be too slow to do in the vector unit. Although the system processor that is associated with a GPU is the closest analogy to a scalar processor in a vector architecture, the separate address spaces plus transferring over a PCIe bus means thousands of clock cycles of overhead to use them together. The scalar processor can be slower than a vector processor for floating-point computations in a vector computer, but not by the same ratio as the system processor versus a multithreaded SIMD Processor (given the overhead).

Therefore each “vector unit” in a GPU must do computations that you would expect to do using a scalar processor in a vector computer. That is, rather than calculate on the system processor and communicate the results, it can be faster to disable all but one SIMD Lane using the predicate registers and built-in masks and do the scalar work with one SIMD Lane. The relatively simple scalar processor in a vector computer is likely to be faster and more power-efficient than the GPU solution. If system processors and GPUs become more closely tied together in the future, it will be interesting to see if system processors can play the same role as scalar processors do for vector and multimedia SIMD architectures.

Similarities and Differences Between Multimedia SIMD Computers and GPUs

At a high level, multicore computers with multimedia SIMD instruction extensions do share similarities with GPUs. Figure 4.23 summarizes the similarities and differences.

Both are multiprocessors whose processors use multiple SIMD Lanes, although GPUs have more processors and many more lanes. Both use hardware multithreading to improve processor utilization, although GPUs have hardware support for many more threads. Both have roughly 2:1 performance ratios between peak performance of single-precision and double-precision floating-point arithmetic. Both use caches, although GPUs use smaller streaming caches, and multicore computers use large multilevel caches that try to contain whole working sets completely. Both use a 64-bit address space, although the physical main memory is much smaller in GPUs. Both support memory protection at the page level as well as demand paging, which allows them to address far more memory than they have on board.

In addition to the large numerical differences in processors, SIMD Lanes, hardware thread support, and cache sizes, there are many architectural differences. The scalar processor and multimedia SIMD instructions are tightly integrated in traditional computers; they are separated by an I/O bus in GPUs, and they even have separate main memories. The multiple SIMD Processors in a GPU use a single address space and can support a coherent view of all memory on some systems given support from CPU vendors (such as the IBM Power9). Unlike GPUs, multimedia SIMD instructions historically did not support gather-scatter memory accesses, which Section 4.7 shows is a significant omission.

Feature	**Multicore with SIMD**	**GPU**
SIMD Processors	4–8	8–32
SIMD Lanes/Processor	2–4	up to 64
Multithreading hardware support for SIMD Threads	2–4	up to 64
Typical ratio of single-precision to double-precision performance	2:1	2:1
Largest cache size	40 MB	4 MB
Size of memory address	64-bit	64-bit
Size of main memory	up to 1024 GB	up to 24 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	Yes
Integrated scalar processor/SIMD Processor	Yes	No
Cache coherent	Yes	Yes on some systems

Figure 4.23 Similarities and differences between multicore with multimedia SIMD extensions and recent GPUs.

Summary

Now that the veil has been lifted, we can see that GPUs are really just multithreaded SIMD Processors, although they have more processors, more lanes per processor, and more multithreading hardware than do traditional multicore computers. For example, the Pascal P100 GPU has 56 SIMD Processors with 64 lanes per processor and hardware support for 64 SIMD Threads. Pascal embraces instruction-level parallelism by issuing instructions from two SIMD Threads to two sets of SIMD Lanes. GPUs also have less cache memory—Pascal's L2 cache is 4 MiB—and it can be coherent with a cooperative distant scalar processor or distant GPUs.

The CUDA programming model wraps up all these forms of parallelism around a single abstraction, the CUDA Thread. Thus the CUDA programmer can think of programming thousands of threads, although they are really executing each block of 32 threads on the many lanes of the many SIMD Processors. The CUDA programmer who wants good performance keeps in mind that these threads are organized in blocks and executed 32 at a time and that addresses need to be to adjacent addresses to get good performance from the memory system.

Although we've used CUDA and the NVIDIA GPU in this section, rest assured that the same ideas are found in the OpenCL programming language and in GPUs from other companies.

Now that you understand better how GPUs work, we reveal the real jargon. Figures 4.24 and 4.25 match the descriptive terms and definitions of this section with the official CUDA/NVIDIA and AMD terms and definitions. We also include the OpenCL terms. We believe the GPU learning curve is steep in part because of using terms such as "streaming multiprocessor" for the SIMD Processor, "thread processor" for the SIMD Lane, and "shared memory" for local memory—especially because local memory is *not* shared between SIMD Processors! We hope that this two-step approach gets you up that curve quicker, even if it's a bit indirect.

4.5 Detecting and Enhancing Loop-Level Parallelism

Loops in programs are the fountainhead of many of the types of parallelism we previously discussed here and in Chapter 5. In this section, we discuss compiler technology used for discovering the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques. We define precisely when a loop is parallel (or vectorizable), how a dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependences. Finding and manipulating loop-level parallelism is critical to exploiting both DLP and TLP, as well as the more aggressive static ILP approaches (e.g., VLIW) that we examine in Appendix H.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange"	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via local memory. AMD and OpenCL name is "work group"	A Thread Block is an array of CUDA Threads that execute concurrently and can cooperate and communicate via shared memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid
	Sequence of SIMD Lane operations	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask. AMD and OpenCL call a CUDA Thread a "work item"	A CUDA Thread is a lightweight thread that executes a sequential program and that can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block
Machine object	A thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront"	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction	A PTX instruction specifies an instruction executed by a CUDA Thread

Figure 4.24 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. OpenCL names are given in the book's definitions.

Loop-level parallelism is normally investigated at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by the renaming techniques discussed in Chapter 3.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such dependence is called a *loop-carried dependence*. Most of the examples

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Short explanation and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multiprocessor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes	A streaming multiprocessor (SM) is a multithreaded SIMT/SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes
	Thread Block Scheduler	Giga Thread Engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine"	Distributes and schedules Thread Blocks of a grid to streaming multiprocessors as resources become available
	SIMD Thread scheduler	Warp scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution. AMD name is "Work Group Scheduler"	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane"	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp
Memory hardware	GPU Memory	Global memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "global memory"	Global memory is accessible by all CUDA Threads in any Thread Block in any grid; implemented as a region of DRAM, and may be cached
	Private memory	Local memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "private memory"	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM
	Local memory	Shared memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "local memory." AMD calls it "group memory"	Fast SRAM memory shared by the CUDA Threads composing a Thread Block, and private to that Thread Block. Used for communication among CUDA Threads in a Thread Block at barrier synchronization points
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "registers"	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor

Figure 4.25 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. Note that our descriptive terms "local memory" and "private memory" use the OpenCL terminology. NVIDIA uses SIMT (single-instruction multiple-thread) rather than SIMD to describe a streaming multiprocessor. SIMT is preferred over SIMD because the per-thread branching and control flow are unlike any SIMD machine.

we considered in Chapters 2 and 3 had no loop-carried dependences and thus are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

```
for (i=999; i>=0; i=i-1)
      x[i] = x[i] + s;
```

In this loop, the two uses of x[i] are dependent, but this dependence is within a single iteration and is not loop-carried. There is a loop-carried dependence between successive uses of i in different iterations, but this dependence involves an induction variable that can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 2.2 of Chapter 2, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, a compiler can do this analysis more easily at or near the source level, in contrast to the machine-code level. Let's look at a more complex example.

Example Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that A, B, and C are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure that includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements `S1` and `S2` in the loop?

Answer There are two different dependences:

1. `S1` uses a value computed by `S1` in an earlier iteration, because iteration i computes A[i + 1], which is read in iteration i + 1. The same is true of `S2` for B[i] and B[i + 1].
2. `S2` uses the value A[i + 1] computed by `S1` in the same iteration.

These two dependences are distinct and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement `S1` is on an earlier iteration of `S1`, this dependence is loop-carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence (`S2` depending on `S1`) is within an iteration and is not loop-carried. Thus, if this were the only dependence, multiple iterations of the loop would execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 2.2, where unrolling could expose the parallelism. These intra-loop dependences are common; for example, a sequence of vector instructions that uses chaining exhibits exactly this sort of dependence.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

Example Consider a loop like this one:

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between `S1` and `S2`? Is this loop parallel? If not, show how to make it parallel.

Answer Statement S1 uses the value assigned in the previous iteration by statement `S2`, so there is a loop-carried dependence between `S2` and `S1`. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although `S1` depends on `S2`, `S2` does not depend on `S1`. A loop is parallel if it can be written without a cycle in the dependences because the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the preceding loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from `S1` to `S2`. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Because this other dependence is absent, interchanging the two statements will not affect the execution of `S2`.
2. On the first iteration of the loop, statement `S2` depends on the value of B[0] computed *prior* to initiating the loop.

These two observations allow us to replace the preceding loop with the following code sequence:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

The dependence between the two statements is no longer loop-carried so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact*, in the sense that it tells us that such dependence *may* exist. Consider the following example:

```
for (i=0;i<100;i=i+1){
      A[i] = B[i] + C[i]
      D[i] = A[i] * E[i]
}
```

The second reference to A in this example need not be translated to a load instruction because we know that the value is computed and stored by the previous statement. Thus the second reference to A can simply be a reference to the register into which A was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis tells that only one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the preceding example, a simple version of this analysis suffices because the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence*. A recurrence occurs when a variable is defined based on the value of that variable in an earlier iteration, usually the one immediately preceding, as in the following code fragment:

```
for (i=1;i<100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

Detecting a recurrence can be important for two reasons: some architectures (especially vector computers) have special support for executing recurrences, and in an ILP context, it may still be possible to exploit a fair amount of parallelism.

Finding Dependences

Clearly, finding the dependences in a program is important both to determine which loops might contain parallelism and to eliminate name dependences. The complexity of dependence analysis arises also because of the presence of arrays and pointers in languages such as C or C++, or pass-by-reference parameter passing in Fortran. Because scalar variable references explicitly refer to a name, they can usually be analyzed quite easily with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine*. In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form x[y[i]], are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the identical value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, that are both within the limits of the for-loop. That is, $m \leq j \leq n$, $m \leq k \leq n$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$, that is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether dependence exists at compile time. For example, the values of a, b, c, and d may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time; for example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where a, b, c, and d are all constants. For these cases, it is possible to devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (GCD) test. It is based on the observation that if a loop-carried dependence exists, then GCD (c, a) must divide ($d - b$). (Recall that an integer, x, *divides* another integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.)

Example Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=0; i<100; i=i+1) {
     X[2*i+3] = X[2*i] * 5.0;
}
```

Answer Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a, c) = 2, and $d - b = -3$. Because 2 does not divide -3, no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not consider the loop bounds.

In general, determining whether a dependence actually exists is NP-complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP-complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

Example The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {
    Y[i] = X[i] / c;  /* S1 */
    X[i] = X[i] + c;  /* S2 */
    Z[i] = Y[i] + c;  /* S3 */
    Y[i] = c - Y[i];  /* S4 */
}
```

Answer The following dependences exist among the four statements:

1. There are true dependences from `S1` to `S3` and from `S1` to `S4` because of Y[i]. These are not loop-carried, so they do not prevent the loop from being considered parallel. These dependences will force `S3` and `S4` to wait for S1 to complete.
2. There is an antidependence from `S1` to `S2`, based on X[i].
3. There is an antidependence from `S3` to `S4` for Y[i].
4. There is an output dependence from `S1` to `S4`, based on Y[i].

The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=0; i<100; i=i+1 {
    T[i] = X[i] / c; /* Y renamed to T to remove output
dependence */
    X1[i] = X[i] + c;/* X renamed to X1 to remove
antidependence */
    Z[i] = T[i] + c;/* Y renamed to T to remove
antidependence */
    Y[i] = c - T[i];
}
```

After the loop, the variable X has been renamed X1. In code that follows the loop, the compiler can simply replace the name X by X1. In this case, renaming does not require an actual copy operation, as it can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

Dependence analysis is a critical technology for exploiting parallelism, as well as for the transformation-like blocking that Chapter 2 covers. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs for vector computers, SIMD computers, or multiprocessors depends critically on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances, namely, among references within a single loop nest and using affine index functions. Thus there are many situations where array-oriented dependence analysis *cannot* tell us what we want to know; for example, analyzing accesses done with pointers, rather than with array indices can be much harder. (This is one reason why Fortran is still preferred over C and C++ for many scientific applications designed for parallel computers.) Similarly, analyzing references across procedure calls is extremely difficult. Thus, while analysis of code written in sequential languages remains important, we also need approaches such as OpenMP and CUDA that write explicitly parallel loops.

Eliminating Dependent Computations

As previously mentioned, one of the most important forms of dependent computations is a recurrence. A dot product is a perfect example of a recurrence:

```
for (i=9999; i>=0; i=i-1)
      sum  =  sum + x[i] * y[i];
```

This loop is not parallel because it has a loop-carried dependence on the variable `sum`. We can, however, transform it to a set of loops, one of which is completely parallel and the other partly parallel. The first loop will execute the completely parallel portion of this loop. It looks like this:

```
for (i=9999;  i>=0;  i=i-1)
      sum[i]  =  x[i]  *  y[i];
```

Notice that sum has been expanded from a scalar into a vector quantity (a transformation called *scalar expansion*) and that this transformation makes this new loop completely parallel. When we are done, however, we need to do the reduce step, which sums up the elements of the vector. It looks like this:

```
for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

Although this loop is not parallel, it has a very specific structure called a *reduction*. Reductions are common in linear algebra, and as we will see in Chapter 6, they are also a key part of the primary parallelism primitive MapReduce used in warehouse-scale computers. In general, any function can be used as a reduction operator, and common cases include operators such as max and min.

Reductions are sometimes handled by special hardware in a vector and SIMD architecture that allows the reduce step to be done much faster than it could be done in scalar mode. These work by implementing a technique similar to what can be done in a multiprocessor environment. While the general transformation works with any number of processors, suppose for simplicity we have 10 processors. In the first step of reducing the sum, each processor executes the following (with `p` as the processor number ranging from 0 to 9):

```
for (i=999; i>=0; i=i-1)
     finalsum[p] = finalsum[p] + sum[i+1000*p];
```

This loop, which sums up 1000 elements on each of the 10 processors, is completely parallel. A simple scalar loop can then complete the summation of the last 10 sums. Similar approaches are used in vector processors and SIMD Processors.

It is important to observe that the preceding transformation relies on associativity of addition. Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, for either integer arithmetic, because of limited range, or floating-point arithmetic, because of both range and precision. Thus using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

4.6 Cross-Cutting Issues

Energy and DLP: Slow and Wide Versus Fast and Narrow

A fundamental power advantage of data-level parallel architectures comes from the energy equation in Chapter 1. Assuming ample data-level parallelism, the performance is the same if we halve the clock rate and double the execution resources: twice the number of lanes for a vector computer, wider registers and ALUs for multimedia SIMD, and more SIMD Lanes for GPUs. If we can lower the voltage while dropping the clock rate, we can actually reduce energy as well as the power for the computation while maintaining the same peak performance. Thus GPUs tend to have lower clock rates than system processors, which rely on high clock rates for performance (see Section 4.7).

Compared to out-of-order processors, DLP processors can have simpler control logic to launch a large number of operations per clock cycle; for example, the control is identical for all lanes in vector processors, and there is no logic to decide on multiple instruction issues or speculative execution logic. They also fetch and decode far fewer instructions. Vector architectures can also make it easier to turn off unused portions of the chip. Each vector instruction explicitly describes all the resources it needs for a number of cycles when the instruction issues.

Banked Memory and Graphics Memory

Section 4.2 noted the importance of substantial memory bandwidth for vector architectures to support unit stride, nonunit stride, and gather-scatter accesses.

To achieve the highest memory performance, stacked DRAMs are used in the top-end GPUs from AMD and NVIDIA. Intel also uses stacked DRAM in its Xeon Phi product. Also known as *high bandwidth memory* (*HBM*, *HBM2*), the memory chips are stacked and placed in the same package as the processing chip. The extensive width (typically 1024–4096 data wires) provides high bandwidth, while placing the memory chips in the same package as the processor chip reduces latency and power consumption. The capacity of stacked DRAM is typically 8–32 GB.

Given all the potential demands on the memory from both the computation tasks and the graphics acceleration tasks, the memory system could see a large number of uncorrelated requests. Unfortunately, this diversity hurts memory performance. To cope, the GPU's memory controller maintains separate queues of traffic bound for different banks, waiting until there is enough traffic to justify opening a row and transferring all requested data at once. This delay improves bandwidth but stretches latency, and the controller must ensure that no processing units starve while waiting for data, for otherwise neighboring processors could become idle. Section 4.7 shows that gather-scatter techniques and memory-bank-aware access techniques can deliver substantial increases in performance versus conventional cache-based architectures.

Strided Accesses and TLB Misses

One problem with strided accesses is how they interact with the translation lookaside buffer (TLB) for virtual memory in vector architectures or GPUs. (GPUs also use TLBs for memory mapping.) Depending on how the TLB is organized and the size of the array being accessed in memory, it is even possible to get one TLB miss for every access to an element in the array! The same type of collision can happen with caches, but the performance impact is probably less.

4.7 Putting It All Together: Embedded Versus Server GPUs and Tesla Versus Core i7

Given the popularity of graphics applications, GPUs are now found in both mobile clients and traditional servers and heavy-duty desktop computers. Figure 4.26 lists the key characteristics of the NVIDIA Tegra Parker system on a chip for embedded clients, which is popular in automobiles, and the Pascal GPU for servers. GPU server engineers hope to be able to do live animation within five years after a movie is released. GPU-embedded engineers in turn want to do what a server or game console does today on their hardware within five more years.

The NVIDIA Tegra P1 has six ARMv8 cores and a smaller Pascal GPU (capable of 750 GFLOPS) and 50 GB/s of memory bandwidth. It is the key component

	NVIDIA Tegra 2	NVIDIA Tesla P100
Market	Automotive, Embedded, Console, Tablet	Desktop, server
System processor	Six-Core ARM (2 Denver2 +4 A57)	Not applicable
System interface	Not applicable	PCI Express × 16 Gen 3
System interface bandwidth	Not applicable	16 GB/s (each direction), 32 GB/s (total)
Clock rate	1.5 GHz	1.4 GHz
SIMD multiprocessors	2	56
SIMD Lanes/SIMD multiprocessor	128	64
Memory interface	128-bit LP-DDR4	4096-bit HBM2
Memory bandwidth	50 GB/s	732 GB/s
Memory capacity	up to 16 GB	up to 16 GB
Transistors	7 billion	15.3 billion
Process	TSMC 16 nm FinFET	TSMC 16 nm FinFET
Die area	147 mm^2	645 mm^2
Power	20 W	300 W

Figure 4.26 Key features of the GPUs for embedded clients and servers.

of the NVIDIA DRIVE PX2 computing platform that is used in cars for autonomous driving. The NVIDIA Tegra X1 is the previous generation and is used in several high-end tablets, such as the Google Pixel C and the NVIDIA Shield TV. It has a Maxwell-class GPU capable of 512 GFLOPS.

The NVIDIA Tesla P100 is the Pascal GPU discussed extensively in this chapter. (Tesla is Nvidia's name for products targeting general-purpose computing.) The clock rate is 1.4 GHz, and it includes 56 SIMD Processors. The path to HBM2 memory is 4096-bits wide, and it transfers data on both the rising and falling edge of a 0.715 GHz clock, which means a peak memory bandwidth of 732 GB/s. It connects to the host system processor and memory via a PCI Express × 16 Gen 3 link, which has a peak bidirectional rate of 32 GB/s.

All physical characteristics of the P100 die are impressively large: it contains 15.3 billion transistors, the die size is 645 mm^2 in a 16-nm TSMC process, and the typical power is 300 W.

Comparison of a GPU and a MIMD With Multimedia SIMD

A group of Intel researchers published a paper (Lee et al., 2010) comparing a quad-core Intel i7 with multimedia SIMD extensions to the Tesla GTX 280. Although the study did not compare the latest versions of CPUs and GPUs, it was the most

	Core i7-960	GTX 280	Ratio 280/i7
Number of processing elements (cores or SMs)	4	30	7.5
Clock frequency (GHz)	3.2	1.3	0.41
Die size	263	576	2.2
Technology	Intel 45 nm	TSMC 65 nm	1.6
Power (chip, not module)	130	130	1.0
Transistors	700 M	1400 M	2.0
Memory bandwidth (GB/s)	32	141	4.4
Single-precision SIMD width	4	8	2.0
Double-precision SIMD width	2	1	0.5
Peak single-precision scalar FLOPS (GFLOP/S)	26	117	4.6
Peak single-precision SIMD FLOPS (GFLOP/S)	102	311–933	3.0–9.1
(SP 1 add or multiply)	N.A.	(311)	(3.0)
(SP 1 instruction fused multiply-adds)	N.A.	(622)	(6.1)
(Rare SP dual issue fused multiply-add and multiply)	N.A.	(933)	(9.1)
Peak double-precision SIMD FLOPS (GFLOP/S)	51	78	1.5

Figure 4.27 Intel Core i7-960 and NVIDIA GTX 280. The rightmost column shows the ratios of GTX 280 to Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program. From Table 2 in Lee, W.V., et al., 2010. Debunking the 100 × GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

in-depth comparison of the two styles in that it explained the reasons behind the differences in performance. Moreover, the current versions of these architectures share many similarities to the ones in the study.

Figure 4.27 lists the characteristics of the two systems. Both products were purchased in the fall of 2009. The Core i7 is in Intel's 45-nanometer semiconductor technology, while the GPU is in TSMC's 65-nanometer technology. Although it might have been fairer to have a comparison done by a neutral party or by both interested parties, the purpose of this section is *not* to determine how much faster one product is than the other, but to try to understand the relative value of features of these two contrasting architecture styles.

The rooflines of the Core i7 920 and GTX 280 in Figure 4.28 illustrate the differences in the computers. The 920 has a slower clock rate than the 960 (2.66 GHz vs. 3.2 GHz), but the rest of the system is the same. Not only does the GTX 280 have much higher memory bandwidth and double-precision floating-point performance, but also its double-precision ridge point is considerably to the left. As previously mentioned, it is much easier to hit peak computational performance the further the ridge point of the roofline is to the left. The double-precision ridge point

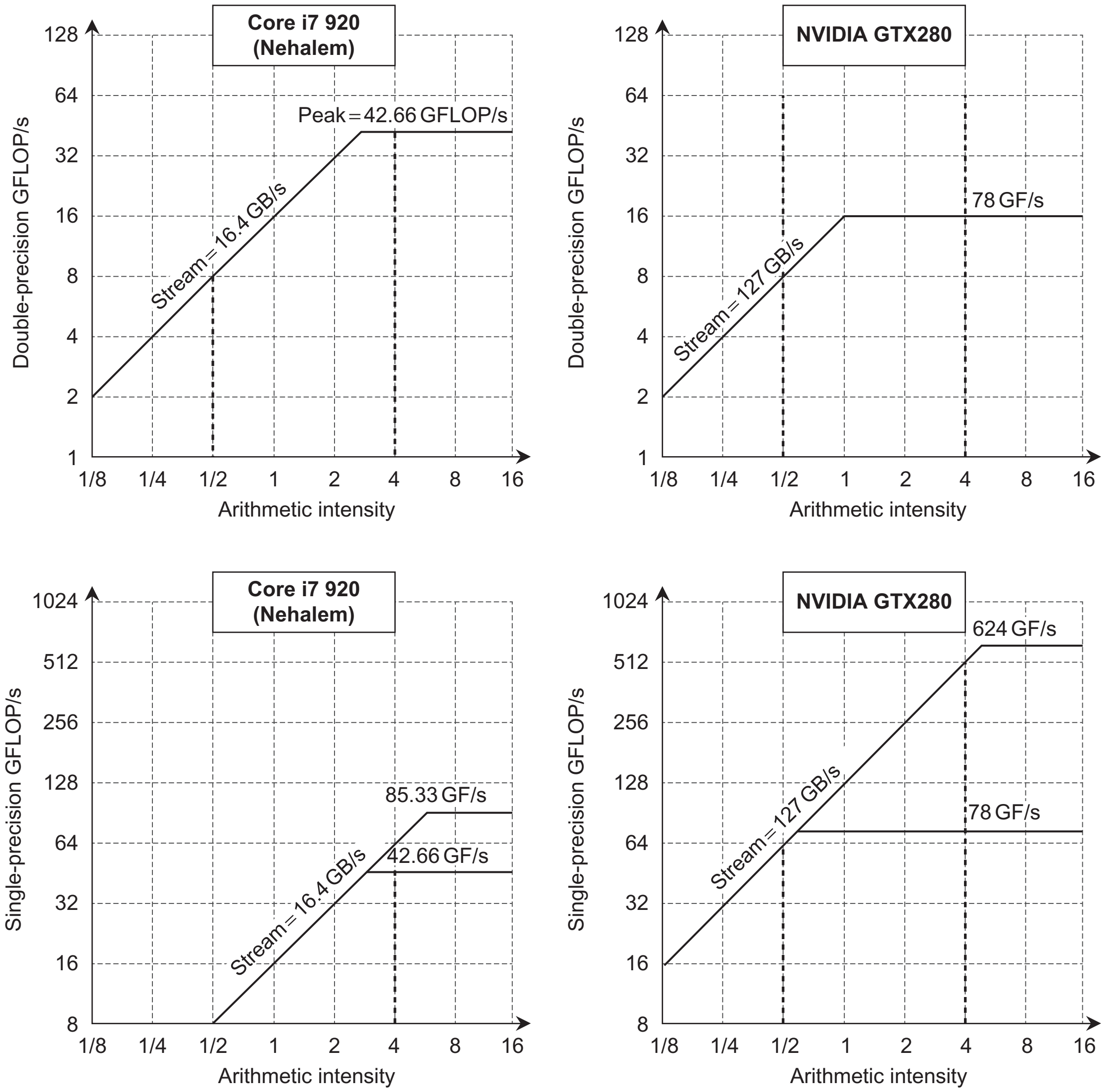


Figure 4.28 Roofline model (Williams et al. 2009). These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/s, a SP FP peak of 85.33 GFLOP/s, and a peak memory bandwidth of 16.4 GB/s. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/s, SP FP peak of 624 GFLOP/s, and 127 GB/s of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/s or 8 SP GFLOP/s on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 42.66 DP GFLOP/s and 64 SP GFLOP/s on the Core i7 and to 78 DP GFLOP/s and 512 DP GFLOP/s on the GTX 280. To hit the highest computation rate on the Core i7, you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD Processors.

is 0.6 for the GTX 280 versus 2.6 for the Core i7. For single-precision performance, the ridge point moves far to the right, as it's considerably harder to hit the roof of single-precision performance because it is so much higher. Note that the arithmetic intensity of the kernel is based on the bytes that go to main memory, not the bytes that go to cache memory. Thus caching can change the arithmetic intensity of a kernel on a particular computer, presuming that most references really go to the cache. The Rooflines help explain the relative performance in this case study. Note also that this bandwidth is for unit-stride accesses in both architectures. Real gather-scatter addresses that are not coalesced are slower on the GTX 280 and on the Core i7, as we will see.

The researchers said that they selected the benchmark programs by analyzing the computational and memory characteristics of four recently proposed benchmark suites and then "formulated the set of *throughput computing kernels* that capture these characteristics." Figure 4.29 describes these 14 kernels, and Figure 4.30 shows the performance results, with larger numbers meaning faster.

Given that the raw performance specifications of the GTX 280 vary from 2.5 × slower (clock rate) to 7.5 × faster (cores per chip) while the performance varies from 2.0 × slower (Solv) to 15.2 × faster (GJK), the Intel researchers explored the reasons for the differences:

- *Memory bandwidth*. The GPU has 4.4 × the memory bandwidth, which helps explain why LBM and SAXPY run 5.0 and 5.3 × faster; their working sets are hundreds of megabytes and thus don't fit into the Core i7 cache. (To access memory intensively, they did not use cache blocking on SAXPY.) Thus the slope of the rooflines explains their performance. SpMV also has a large working set, but it only runs 1.9 × because the double-precision floating point of the GTX 280 is just 1.5 × faster than the Core i7.
- *Compute bandwidth*. Five of the remaining kernels are compute bound: SGEMM, Conv, FFT, MC, and Bilat. The GTX is faster by 3.9, 2.8, 3.0, 1.8, and 5.7, respectively. The first three of these use single-precision floating-point arithmetic, and GTX 280 single-precision is 3–6 × faster. (The 9 × faster than the Core i7 as shown in Figure 4.27 occurs only in the very special case when the GTX 280 can issue a fused multiply-add and a multiply per clock cycle.) MC uses double-precision, which explains why it's just 1.8 × faster since DP performance is only 1.5 × faster. Bilat uses transcendental functions, which the GTX 280 supports directly (see Figure 4.17). The Core i7 spends two-thirds of its time calculating transcendental functions, so the GTX 280 is 5.7 × faster. This observation helps point out the value of hardware support for operations that occur in your workload: double-precision floating-point and perhaps even transcendentals.
- *Cache benefits*. Ray casting (RC) is only 1.6 × faster on the GTX because cache blocking with the Core i7 caches prevents it from becoming memory bandwidth bound, as it is on GPUs. Cache blocking can help Search, too. If the index trees are small so that they fit into the cache, the Core i7 is twice

Kernel	Application	SIMD	TLP	Characteristics
SGEMM (**SGEMM**)	Linear algebra	Regular	Across 2D tiles	Compute bound after tiling
Monte Carlo (**MC**)	Computational finance	Regular	Across paths	Compute bound
Convolution (**Conv**)	Image analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT (**FFT**)	Signal processing	Regular	Across smaller FFTs	Compute bound or BW bound depending on size
SAXPY (**SAXPY**)	Dot product	Regular	Across vector	BW bound for large vectors
LBM (**LBM**)	Time migration	Regular	Across cells	BW bound
Constraint solver (**Solv**)	Rigid body physics	Gather/ Scatter	Across constraints	Synchronization bound
SpMV (**SpMV**)	Sparse solver	Gather	Across nonzero	BW bound for typical large matrices
GJK (**GJK**)	Collision detection	Gather/ Scatter	Across objects	Compute bound
Sort (**Sort**)	Database	Gather/ Scatter	Across elements	Compute bound
Ray casting (**RC**)	Volume rendering	Gather	Across rays	4–8 MB first level working set; over 500 MB last level working set
Search (**Search**)	Database	Gather/ Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram (**Hist**)	Image analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound
Bilateral (**Bilat**)	Image analysis	Regular	Across pixels	Compute bound

Figure 4.29 Throughput computing kernel characteristics. The name in parentheses identifies the benchmark name in this section. The authors suggest that code for both machines had equal optimization effort. From Table 1 in Lee, W.V., et al., 2010. Debunking the 100 × GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

as fast. Larger index trees make them memory bandwidth bound. Overall, the GTX 280 runs Search 1.8 × faster. Cache blocking also helps Sort. While most programmers wouldn't run Sort on a SIMD Processor, it can be written with a 1-bit Sort primitive called *split*. However, the split algorithm executes many more instructions than a scalar sort does. As a result, the GTX 280 runs only 0.8 × as fast as the Core i7. Note that caches also help other kernels on the Core i7 because cache blocking allows SGEMM, FFT, and SpMV to become

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/s	94	364	3.9
MC	Billion paths/s	0.8	1.4	1.8
Conv	Million pixels/s	1250	3500	2.8
FFT	GFLOP/s	71.4	213	3.0
SAXPY	GB/s	16.8	88.8	5.3
LBM	Million lookups/s	85	426	5.0
Solv	Frames/s	103	52	0.5
SpMV	GFLOP/s	4.9	9.1	1.9
GJK	Frames/s	67	1020	15.2
Sort	Million elements/s	250	198	0.8
RC	Frames/s	5	8.1	1.6
Search	Million queries/s	50	90	1.8
Hist	Million pixels/s	1517	2583	1.7
Bilat	Million pixels/s	83	475	5.7

Figure 4.30 Raw and relative performance measured for the two platforms. In this study, SAXPY is used only as a measure of memory bandwidth, so the right unit is GB/s and not GFLOP/s. Based on Table 3 in Lee, W.V., et al., 2010. Debunking the 100× GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

compute bound. This observation reemphasizes the importance of cache blocking optimizations in Chapter 2.

- *Gather-Scatter*. The multimedia SIMD extensions are of little help if the data are scattered throughout main memory; optimal performance comes only when data are aligned on 16-byte boundaries. Thus GJK gets little benefit from SIMD on the Core i7. As previously mentioned, GPUs offer gather-scatter addressing that is found in a vector architecture but omitted from SIMD extensions. The Address Coalescing Unit helps as well by combining accesses to the same DRAM line, thereby reducing the number of gathers and scatters. The memory controller also batches together accesses to the identical DRAM page. This combination means the GTX 280 runs GJK a startling 15.2× faster than the Core i7, which is larger than any single physical parameter in Figure 4.27. This observation reinforces the importance of gather-scatter to vector and GPU architectures that is missing from SIMD extensions.
- *Synchronization*. The performance synchronization of Hist is limited by atomic updates, which are responsible for 28% of the total runtime on the Core i7 despite its having a hardware fetch-and-increment instruction. Thus Hist is

> only 1.7 × faster on the GTX 280. Solv solves a batch of independent constraints in a small amount of computation followed by barrier synchronization. The Core i7 benefits from the atomic instructions and a memory consistency model that ensures the right results even if not all previous accesses to memory hierarchy have completed. Without the memory consistency model, the GTX 280 version launches some batches from the system processor, which leads to the GTX 280 running 0.5 × as fast as the Core i7. This observation points out how synchronization performance can be important for some data parallel problems.

It was interesting that the gather-scatter support of vector architectures, which predate the SIMD instructions by decades, was so important to the effective usefulness of these SIMD extensions, which some had predicted before the comparison (Gebis and Patterson, 2007). The Intel researchers noted that 6 of the 14 kernels would exploit SIMD better with more efficient gather-scatter support on the Core i7.

Note that an important feature missing from this comparison was describing the level of effort to get the results for the two systems. Ideally, future comparisons would release the code used on both systems so that others could re-create the same experiments on different hardware platforms and possibly improve on the results.

Comparison Update

In the intervening years, the weaknesses of the Core i7 and Tesla GTX 280 have been addressed by their successors. Intel's ACV2 added gather instructions, and AVX/512 added scatter instructions, both of which are found in the Intel Skylake series. Nvidia Pascal has double-precision floating-point performance that is one-half instead of one-eighth the speed of single precision, fast atomic operations, and caches.

Figure 4.31 lists the characteristics of these two successors, Figure 4.32 compares performance using 3 of the 14 benchmarks in the original paper (those were the ones for which we could find source code), and Figure 4.33 shows the two new roofline models. The new GPU chip is 15 to 50 times faster and the new CPU chips is 50 times faster than their predecessors, and the new GPU is 2–5 times faster than the new CPU.

4.8 Fallacies and Pitfalls

While data-level parallelism is the easiest form of parallelism after ILP from the programmer's perspective, and plausibly the simplest from the architect's perspective, it still has many fallacies and pitfalls.

Fallacy *GPUs suffer from being coprocessors.*

Although the split between main memory and GPU memory has disadvantages, there are advantages to being at a distance from the CPU.

	Xeon Platinum 8180	P100	Ratio P100/Xeon
Number of processing elements (cores or SMs)	28	56	2.0
Clock frequency (GHz)	2.5	1.3	0.52
Die size	N.A.	610 mm^2	–
Technology	Intel 14 nm	TSMC 16 nm	1.1
Power (chip, not module)	80 W	300 W	3.8
Transistors	N.A.	15.3 B	–
Memory bandwidth (GB/s)	199	732	3.7
Single-precision SIMD width	16	8	0.5
Double-precision SIMD width	8	4	0.5
Peak single-precision SIMD FLOPS (GFLOP/s)	4480	10,608	2.4
Peak double-precision SIMD FLOPS (GFLOP/s)	2240	5304	2.4

Figure 4.31 Intel Xeon ?? and NVIDIA P100. The rightmost column shows the ratios of P100 to the Xeon. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program.

Kernel	Units	Xeon Platinum 8180	P100	P100/Xeon	GTX 280/i7-960
SGEMM	GFLOP/s	3494	6827	2.0	3.9
DGEMM	GFLOP/s	1693	3490	2.1	—
FFT-S	GFLOP/s	410	1820	4.4	3.0
FFT-D	GFLOP/s	190	811	4.2	—
SAXPY	GB/s	207	544	2.6	5.3
DAXPY	GB/s	212	556	2.6	—

Figure 4.32 Raw and relative performance measured for modern versions of the two platforms as compared to the relative performance of the original platforms. Like Figure 4.30, SAXPY and DAXPY are used only as a measure of memory bandwidth, so the proper unit is GB/s and not GFLOP/s.

For example, PTX exists in part because of the I/O device nature of GPUs. This level of indirection between the compiler and the hardware gives GPU architects much more flexibility than system processor architects. It's often hard to know in advance whether an architecture innovation will be well supported by compilers and libraries and be important to applications. Sometimes a new mechanism will even prove useful for one or two generations and then fade in importance as the IT world changes. PTX allows GPU architects to try innovations speculatively and drop them in subsequent generations if they disappoint or fade in importance, which encourages experimentation. The justification for inclusion is understandably considerably higher for system processors—and thus much less

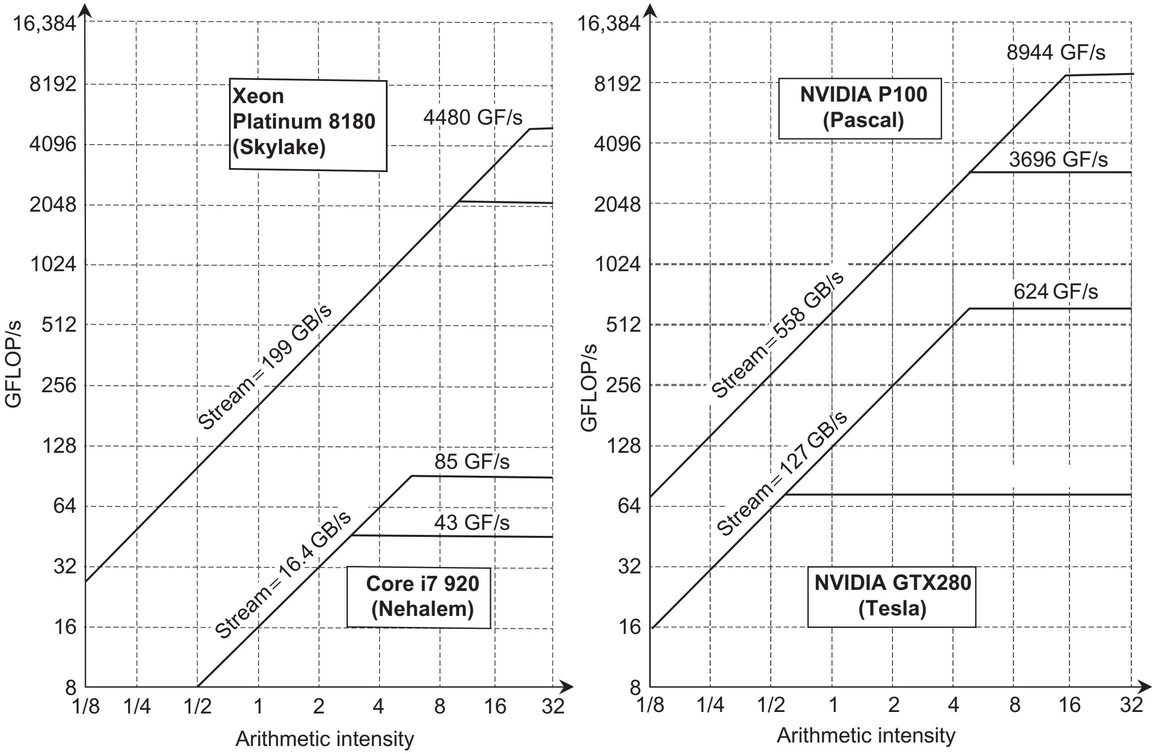


Figure 4.33 Roofline models of older and newer CPUs versus older and newer GPUs. The higher roofline for each computer is single-precision floating-point performance, and the lower one is double-precision performance.

experimentation can occur—as distributing binary machine code normally implies that new features must be supported by all future generations of that architecture.

A demonstration of the value of PTX is that the different generation architecture radically changed the hardware instruction set—from being memory-oriented like x86 to being register-oriented like RISC-V *as well as* doubling the address size to 64 bits—without disrupting the NVIDIA software stack.

Pitfall *Concentrating on peak performance in vector architectures and ignoring start-up overhead.*

Early memory-memory vector processors such as the TI ASC and the CDC STAR-100 had long start-up times. For some vector problems, vectors had to be longer than 100 for the vector code to be faster than the scalar code! On the CYBER 205—derived from the STAR-100—the start-up overhead for DAXPY is 158 clock cycles, which substantially increases the break-even point. If the clock rates of the Cray-1 and the CYBER 205 were identical, the Cray-1 would be faster until the vector length was greater than 64. Because the Cray-1 clock rate was also higher (even though the 205 was newer), the crossover point was a vector length over 100.

Pitfall *Increasing vector performance, without comparable increases in scalar performance.*

This imbalance was a problem on many early vector processors, and a place where Seymour Cray (the architect of the Cray computers) rewrote the rules. Many of the early vector processors had comparatively slow scalar units (as well as large start-up overheads). Even today, a processor with lower vector performance but better scalar performance can outperform a processor with higher peak vector performance. Good scalar performance keeps down overhead costs (strip mining, for example) and reduces the impact of Amdahl's law.

An excellent example of this comes from comparing a fast scalar processor and a vector processor with lower scalar performance. The Livermore Fortran kernels are a collection of 24 scientific kernels with varying degrees of vectorization. Figure 4.34 shows the performance of two different processors on this benchmark. Despite the vector processor's higher peak performance, its low scalar performance makes it slower than a fast scalar processor as measured by the harmonic mean.

The flip of this danger today is increasing vector performance—say, by increasing the number of lanes—without increasing scalar performance. Such myopia is another path to an unbalanced computer.

The next fallacy is closely related.

Fallacy *You can get good vector performance without providing memory bandwidth.*

As we saw with the DAXPY loop and the Roofline model, memory bandwidth is quite important to all SIMD architectures. DAXPY requires 1.5 memory references per floating-point operation, and this ratio is typical of many scientific codes. Even if the floating-point operations took no time, a Cray-1 could not increase the performance of the vector sequence used, because it is memory-limited. The Cray-1 performance on Linpack jumped when the compiler used blocking to change the computation so that values could be kept in the vector registers. This approach lowered the number of memory references per FLOP and improved the performance by nearly a factor of two! Thus the memory bandwidth on the Cray-1 became sufficient for a loop that formerly required more bandwidth.

Processor	Minimum rate for any loop (MFLOPS)	Maximum rate for any loop (MFLOPS)	Harmonic mean of all 24 loops (MFLOPS)
MIPS M/120-5	0.80	3.89	1.85
Stardent-1500	0.41	10.08	1.72

Figure 4.34 Performance measurements for the Livermore Fortran kernels on two different processors. Both the MIPS M/120-5 and the Stardent-1500 (formerly the Ardent Titan-1) use a 16.7 MHz MIPS R2000 chip for the main CPU. The Stardent-1500 uses its vector unit for scalar FP and has about half the scalar performance (as measured by the minimum rate) of the MIPS M/120-5, which uses the MIPS R2010 FP chip. The vector processor is more than a factor of 2.5 × faster for a highly vectorizable loop (maximum rate). However, the lower scalar performance of the Stardent-1500 negates the higher vector performance when total performance is measured by the harmonic mean on all 24 loops.

Fallacy *On GPUs, just add more threads if you don't have enough memory performance.*

GPUs use many CUDA Threads to hide the latency to main memory. If memory accesses are scattered or not correlated among CUDA Threads, the memory system will get progressively slower in responding to each individual request. Eventually, even many threads will not cover the latency. For the "more CUDA Threads" strategy to work, not only do you need lots of CUDA Threads, but the CUDA Threads themselves also must be well behaved in terms of locality of memory accesses.

4.9 Concluding Remarks

Data-level parallelism is increasing in importance for personal mobile devices, given the popularity of applications showing the importance of audio, video, and games on these devices. When combined with a model that is easier to program than task-level parallelism and with potentially better energy efficiency, it's easy to see why there has been a renaissance for data-level parallelism in this decade.

We are seeing system processors take on more of the characteristics of GPUs, and vice versa. One of the biggest differences in performance between conventional processors and GPUs has been for gather-scatter addressing. Traditional vector architectures show how to add such addressing to SIMD instructions, and we expect to see more ideas added from the well-proven vector architectures to SIMD extensions over time.

As we said in the opening of Section 4.4, the GPU question is not simply which architecture is best, but given the hardware investment to do graphics well, how can it be enhanced to support computation that is more general? Although vector architectures have many advantages on paper, it remains to be proven whether vector architectures can be as good a foundation for graphics as GPUs. RISC-V has embraced vector over SIMD. Thus, like architecture debates of the past, the marketplace will help determine the importance of the strengths and weaknesses of two styles of data parallel architectures.

4.10 Historical Perspective and References

Section M.6 (available online) features a discussion on the Illiac IV (a representative of the early SIMD architectures) and the Cray-1 (a representative of vector architectures). We also look at multimedia SIMD extensions and the history of GPUs.

Case Study and Exercises by Jason D. Bakos

Case Study: Implementing a Vector Kernel on a Vector Processor and GPU

Concepts illustrated by this case study

- Programming Vector Processors
- Programming GPUs
- Performance Estimation

MrBayes is a popular computational biology application for inferring the evolutionary histories among a set of input species based on their prealigned DNA sequence data of length *n*. MrBayes works by performing an heuristic search over the space of all binary tree topologies for which the inputs are the leaves. In order to evaluate a particular tree, the application must compute an $n \times 4$ conditional likelihood table (named clP) for each interior node. The table is a function of the conditional likelihood tables of the node's two descendent nodes (`clL` and `clR`, single precision floating point) and the 4×4 transition probability table (`tiPL` and `tiPR`, single precision floating point). One of this application's kernels is the computation of this conditional likelihood table and is shown as follows:

```
for (k=0; k<seq_length; k++) {
  clP[h++] = (tiPL[AA]*clL[A] + tiPL[AC]*clL[C] +
               tiPL[AG]*clL[G] + tiPL[AT]*clL[T])*
              (tiPR[AA]*clR[A] + tiPR[AC]*clR[C] +
               tiPR[AG]*clR[G] + tiPR[AT]*clR[T]);
  clP[h++] = (tiPL[CA]*clL[A] + tiPL[CC]*clL[C] +
               tiPL[CG]*clL[G] + tiPL[CT]*clL[T])*
              (tiPR[CA]*clR[A] + tiPR[CC]*clR[C] +
               tiPR[CG]*clR[G] + tiPR[CT]*clR[T]);
  clP[h++] = (tiPL[GA]*clL[A] + tiPL[GC]*clL[C] +
               tiPL[GG]*clL[G] + tiPL[GT]*clL[T])*
              (tiPR[GA]*clR[A] + tiPR[GC]*clR[C] +
               tiPR[GG]*clR[G] + tiPR[GT]*clR[T]);
  clP[h++] = (tiPL[TA]*clL[A] + tiPL[TC]*clL[C] +
               tiPL[TG]*clL[G] + tiPL[TT]*clL[T])*
              (tiPR[TA]*clR[A] + tiPR[TC]*clR[C] +
               tiPR[TG]*clR[G] + tiPR[TT]*clR[T]);
  clL += 4;
  clR += 4;
}
```

4.1 [25] <4.1, 4.2> Assume the constants shown as follows.

Constants	**Values**
AA,AC,AG,AT	0,1,2,3
CA,CC,CG,CT	4,5,6,7
GA,GC,GG,GT	8,9,10,11
TA,TC,TG,TT	12,13,14,15
A,C,G,T	0,1,2,3

Write code for RISC-V and RV64V. Assume the starting addresses of `tiPL`, `tiPR`, `clL`, `clR`, and `clP` are in `RtiPL`, `RtiPR`, `RclL`, `RclR`, and `RclP`, respectively. Do not unroll the loop. To facilitate vector addition reductions, assume that we add the following instructions to RV64V:
Vector Summation Reduction Single Precision:

```
vsum Fd, Vs
```

This instruction performs a summation reduction on a vector register `Vs`, writing to the sum into scalar register `Fd`.

4.2 [5] <4.1, 4.2> Assuming `seq_length` == `500`, what is the dynamic instruction count for both implementations?

4.3 [25] <4.1, 4.2> Assume that the vector reduction instruction is executed on the vector functional unit, similar to a vector add instruction. Show how the code sequence lays out in convoys assuming a single instance of each vector functional unit. How many chimes will the code require? How many cycles per FLOP are needed, ignoring vector instruction issue overhead?

4.4 [15] <4.1, 4.2> Consider the possibility of unrolling the loop and mapping multiple iterations to vector operations. Assume that you can use scatter-gather loads and stores (`vldi` and `vsti`). How does this affect the way you can write the RV64Vcode for this kernel?

4.5 [25] <4.4> Now assume we want to implement the MrBayes kernel on a GPU using a single thread block. Rewrite the C code of the kernel using CUDA. Assume that pointers to the conditional likelihood and transition probability tables are specified as parameters to the kernel. Invoke one thread for each iteration of the loop. Load any reused values into shared memory before performing operations on it.

4.6 [15] <4.4> With CUDA we can use coarse-grain parallelism at the block level to compute the conditional likelihood of multiple nodes in parallel. Assume that we want to compute the conditional likelihood from the bottom of the tree up. Assume seq_length == 500 for all notes and that the group of tables for each of the 12 leaf nodes is stored in consecutive memory locations in the order of node number (e.g., the mth element of clP on node n is at clP [n*4*seq_length + m*4]). Assume that we want to compute the conditional likelihood for nodes 12–17, as shown in Figure 4.35. Change the method by which you compute the array indices in your answer from Exercise 4.5 to include the block number.

4.7 [15] <4.4> Convert your code from Exercise 4.6 into PTX code. How many instructions are needed for the kernel?

4.8 [10] <4.4> How well do you expect this code to perform on a GPU? Explain your answer.

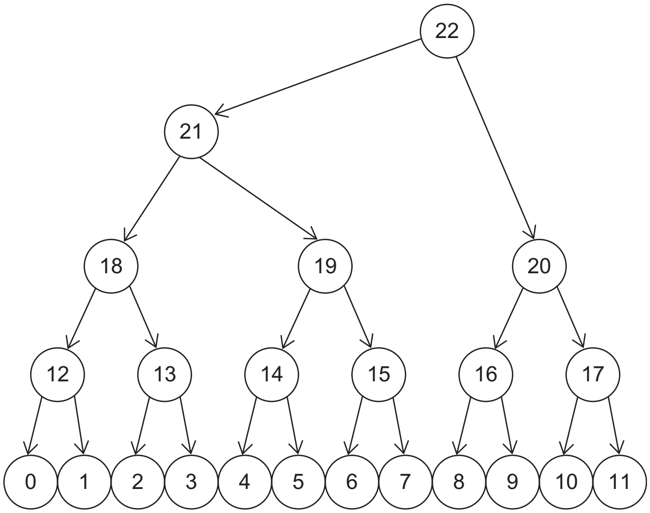


Figure 4.35 Sample tree.

Exercises

4.9 [10/20/20/15/15] <4.2> Consider the following code, which multiplies two vectors that contain single-precision complex values:

```
for (i=0;i <300;i++) {
     c_re[i] = a_re[i] * b_re[i] – a_im[i] * b_im[i];
     c_im[i] = a_re[i] * b_im[i] + a_im[i] * b_re[i];
}
```

Assume that the processor runs at 700 MHz and has a maximum vector length of 64. The load/store unit has a start-up overhead of 15 cycles; the multiply unit, 8 cycles; and the add/subtract unit, 5 cycles.

a. [10] <4.3> What is the arithmetic intensity of this kernel? Justify your answer.

b. [20] <4.2> Convert this loop into RV64V assembly code using strip mining.

c. [20] <4.2> Assuming chaining and a single memory pipeline, how many chimes are required? How many clock cycles are required per complex result value, including start-up overhead?

d. [15] <4.2> If the vector sequence is chained, how many clock cycles are required per complex result value, including overhead?

e. [15] <4.2> Now assume that the processor has three memory pipelines and chaining. If there are no bank conflicts in the loop's accesses, how many clock cycles are required per result?

4.10 [30] <4.2,4.3,4.4> In this problem, we will compare the performance of a vector processor with a hybrid system that contains a scalar processor and a GPU-based coprocessor. In the hybrid system, the host processor has superior scalar performance to the GPU, so in this case all scalar code is executed on the host processor while all vector code is executed on the GPU. We will refer to the first system as the vector computer and the second system as the hybrid computer. Assume that your target application contains a vector kernel with an arithmetic intensity of 0.5 FLOPs per DRAM byte accessed; however, the application also has a scalar component that must be performed before and after the kernel in order to prepare the input vectors and output vectors, respectively. For a sample dataset, the scalar portion of the code requires 400 ms of execution time on both the vector processor and the host processor in the hybrid system. The kernel reads input vectors consisting of 200 MB of data and has output data consisting of 100 MB of data. The vector processor has a peak memory bandwidth of 30 GB/s and the GPU has a peak memory bandwidth of 150 GB/s. The hybrid system has an additional overhead that requires all input vectors to be transferred between the host memory and GPU local memory before and after the kernel is invoked. The hybrid system has a direct memory access (DMA) bandwidth of 10 GB/s and an average latency of 10 ms. Assume that both the vector processor and GPU are performance bound by memory bandwidth. Compute the execution time required by both computers for this application.

4.11 [15/25/25] <4.4, 4.5> Section 4.5 discussed the reduction operation that reduces a vector down to a scalar by repeated application of an operation. A reduction is a special type of a loop recurrence. An example is shown as follows:

```
dot=0.0;
for (i=0;i <64;i++)  dot  =  dot + a[i]  *  b[i];
```

A vectorizing compiler might apply a transformation called *scalar expansion*, which expands dot into a vector and splits the loop such that the multiply can be performed with a vector operation, leaving the reduction as a separate scalar operation:

```
for (i=0;i <64;i++) dot[i]  =  a[i]  *  b[i];
for (i=1;i <64;i++) dot[0]  =  dot[0]  +  dot[i];
```

As mentioned in Section 4.5, if we allow the floating-point addition to be associative, there are several techniques available for parallelizing the reduction.

a. [15] <4.4, 4.5> One technique is called recurrence doubling, which adds sequences of progressively shorter vectors (ie, two 32-element vectors, then two 16-element vectors, and so on). Show how the C code would look for executing the second loop in this way.

b. [25] <4.4, 4.5> In some vector processors, the individual elements within the vector registers are addressable. In this case, the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction called *partial sums*. The idea is to reduce the vector to *m* sums where *m* is the total latency through the vector functional unit, including the operand read and write times. Assume that the VMIPS vector registers are addressable (e.g., you can initiate a vector operation with the operand `V1(16)`, indicating that the input operand begins with element 16). Also, assume that the total latency for adds, including the operand read and result write, is eight cycles. Write a VMIPS code sequence that reduces the contents of `V1` to eight partial sums.

c. [25] <4.4, 4.5> When performing a reduction on a GPU, one thread is associated with each element in the input vector. The first step is for each thread to write its corresponding value into shared memory. Next, each thread enters a loop that adds each pair of input values. This reduces the number of elements by half after each iteration, meaning that the number of active threads also reduces by half after each iteration. In order to maximize the performance of the reduction, the number of fully populated warps should be maximized throughout the course of the loop. In other words, the active threads should be contiguous. Also, each thread should index the shared array in such a way as to avoid bank conflicts in the shared memory. The following loop violates only the first of these guidelines and also uses the modulo operator, which is very expensive for GPUs:

```
unsigned int tid = threadIdx.x;
 for(unsigned int s=1; s <blockDim.x; s * = 2) {
 if ((tid % (2*s)) == 0) {
 sdata[tid] += sdata[tid + s];
 }
 __syncthreads();
 }
```

Rewrite the loop to meet these guidelines and eliminate the use of the modulo operator. Assume that there are 32 threads per warp and a bank conflict occurs whenever two or more threads from the same warp reference an index whose modulo by 32 are equal.

4.12 [10/10/10/10] <4.3> The following kernel performs a portion of the finite-difference time-domain (FDTD) method for computing Maxwell's equations in a three-dimensional space, part of one of the SPEC06fp benchmarks:

```
for (int x=0; x <NX −1; x++) {
 for (int y=0; y <NY −1; y++) {
 for (int z=0; z <NZ −1; z++) {
 int index = x*NY*NZ + y*NZ + z;
 if (y >0 && x >0) {
 material = IDx[index];
 dH1 = (Hz[index] −Hz[index-incrementY])/dy[y];
 dH2 = (Hy[index] −Hy[index-incrementZ])/dz[z];
 Ex[index] = Ca[material]*Ex[index]+Cb[material]*
 (dH2 −dH1);
}}}}
```

Assume that `dH1`, `dH2`, `Hy`, `Hz`, `dy`, `dz`, `Ca`, `Cb`, and `Ex` are all single-precision floating-point arrays. Assume `IDx` is an array of unsigned int.

a. [10] <4.3> What is the arithmetic intensity of this kernel?

b. [10] <4.3> Is this kernel amenable to vector or SIMD execution? Why or why not?

c. [10] <4.3> Assume this kernel is to be executed on a processor that has 30 GB/s of memory bandwidth. Will this kernel be memory bound or compute bound?

d. [10] <4.3> Develop a roofline model for this processor, assuming it has a peak computational throughput of 85 GFLOP/s.

4.13 [10/15] <4.4> Assume a GPU architecture that contains 10 SIMD processors. Each SIMD instruction has a width of 32 and each SIMD processor contains 8 lanes for single-precision arithmetic and load/store instructions, meaning that each nondiverged SIMD instruction can produce 32 results every 4 cycles. Assume a kernel that has divergent branches that causes, on average, 80% of threads to be active. Assume that 70% of all SIMD instructions executed are single-precision arithmetic and 20% are load/store. Because not all memory latencies are covered, assume an average SIMD instruction issue rate of 0.85. Assume that the GPU has a clock speed of 1.5 GHz.

a. [10] <4.4> Compute the throughput, in GFLOP/s, for this kernel on this GPU.

b. [15] <4.4> Assume that you have the following choices:
   (1) Increasing the number of single-precision lanes to 16
   (2) Increasing the number of SIMD processors to 15 (assume this change doesn't affect any other performance metrics and that the code scales to the additional processors)
   (3) Adding a cache that will effectively reduce memory latency by 40%, which will increase instruction issue rate to 0.95

What is speedup in throughput for each of these improvements?

4.14 [10/15/15] <4.5> In this exercise, we will examine several loops and analyze their potential for parallelization.

a. [10] <4.5> Does the following loop have a loop-carried dependency?

```
for (i=0;i <100;i++) {
 A[i] = B[2*i+4];
 B[4*i+5] = A[i];
}
```

b. [15] <4.5> In the following loop, find all the true dependences, output dependences, and antidependences. Eliminate the output dependences and antidependences by renaming.

```
for (i=0;i <100;i++) {
A[i]  =  A[i] * B[i]; /* S1 */
```

```
B[i] = A[i] + c; /* S2 */
A[i] = C[i] * c; /* S3 */
C[i] = D[i] * A[i]; /* S4 */
```

c. [15] <4.5> Consider the following loop:

```
for (i=0;i <100;i++) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

Are there dependences between `S1` and `S2`? Is this loop parallel? If not, show how to make it parallel.

4.15 [10] <4.4> List and describe at least four factors that influence the performance of GPU kernels. In other words, which runtime behaviors that are caused by the kernel code cause a reduction in resource utilization during kernel execution?

4.16 [10] <4.4> Assume a hypothetical GPU with the following characteristics:

- Clock rate 1.5 GHz
- Contains 16 SIMD processors, each containing 16 single-precision floating-point units
- Has 100 GB/s off-chip memory bandwidth

Without considering memory bandwidth, what is the peak single-precision floating-point throughput for this GPU in GLFOP/s, assuming that all memory latencies can be hidden? Is this throughput sustainable given the memory bandwidth limitation?

4.17 [60] <4.4> For this programming exercise, you will write and characterize the behavior of a CUDA kernel that contains a high amount of data-level parallelism but also contains conditional execution behavior. Use the NVIDIA CUDA Toolkit along with GPU-SIM from the University of British Columbia (http://www.gpgpu-sim.org/) or the CUDA Profiler to write and compile a CUDA kernel that performs 100 iterations of Conway's Game of Life for a 256 × 256 game board and returns the final state of the game board to the host. Assume that the board is initialized by the host. Associate one thread with each cell. Make sure you add a barrier after each game iteration. Use the following game rules:

- Any live cell with fewer than two live neighbors dies.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies.
- Any dead cell with exactly three live neighbors becomes a live cell.

After finishing the kernel answer the following questions:

a. [60] <4.4> Compile your code using the –ptx option and inspect the PTX representation of your kernel. How many PTX instructions make up the PTX implementation of your kernel? Did the conditional sections of your kernel include branch instructions or only predicated nonbranch instructions?

b. [60] <4.4> After executing your code in the simulator, what is the dynamic instruction count? What is the achieved instructions per cycle (IPC) or instruction issue rate? What is the dynamic instruction breakdown in terms of control instructions, arithmetic-logical unit (ALU) instructions, and memory instructions? Are there any shared memory bank conflicts? What is the effective off-chip memory bandwidth?

c. [60] <4.4> Implement an improved version of your kernel where off-chip memory references are coalesced and observe the differences in runtime performance.

5.1 Introduction 368
5.2 Centralized Shared-Memory Architectures 377
5.3 Performance of Symmetric Shared-Memory Multiprocessors 393
5.4 Distributed Shared-Memory and Directory-Based Coherence 404
5.5 Synchronization: The Basics 412
5.6 Models of Memory Consistency: An Introduction 417
5.7 Cross-Cutting Issues 422
5.8 Putting It All Together: Multicore Processors and Their Performance 426
5.9 Fallacies and Pitfalls 438
5.10 The Future of Multicore Scaling 442
5.11 Concluding Remarks 444
5.12 Historical Perspectives and References 445
Case Studies and Exercises by Amr Zaky and David A. Wood 446

5

Thread-Level Parallelism

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. . . . Electronic circuits are ultimately limited in their speed of operation by the speed of light . . . and many of the circuits were already operating in the nanosecond range.

W. Jack Bouknight et al.,
The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel's future direction at the Intel Developer Forum in 2005

Since 2004 processor designers have increased core counts to exploit Moore's Law scaling, rather than focusing on single-core performance. The failure of Dennard scaling, to which the shift to multicore parts is partially a response, may soon limit multicore scaling just as single-core scaling has been curtailed.

Hadi Esmaeilzadeh, et al.,
Power Limitations and Dark Silicon Challenge the Future of Multicore (2012)

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00005-5
© 2019 Elsevier Inc. All rights reserved.

5.1 Introduction

As the quotations that open this chapter show, the view that advances in uniprocessor architecture were nearing an end has been held by some researchers for many years. Clearly, these views were premature; in fact, during the period of 1986–2003, uniprocessor performance growth, driven by the microprocessor, was at its highest rate since the first transistorized computers in the late 1950s and early 1960s.

Nonetheless, the importance of multiprocessors was growing throughout the 1990s as designers sought a way to build servers and supercomputers that achieved higher performance than a single microprocessor, while exploiting the tremendous cost-performance advantages of commodity microprocessors. As we discussed in Chapters 1 and 3, the slowdown in uniprocessor performance arising from diminishing returns in exploiting instruction-level parallelism (ILP) combined with growing concern over power has led to a new era in computer architecture—an era where multiprocessors play a major role from the low end to the high end. The second quotation captures this clear inflection point.

This increased importance of multiprocessing reflects several major factors:

- The dramatically lower efficiencies in silicon and energy use that were encountered between 2000 and 2005 as designers attempted to find and exploit more ILP, which turned out to be inefficient, since power and silicon costs grew faster than performance. Other than ILP, the only scalable and general-purpose way we know to increase performance faster than the basic technology allows (from a switching perspective) is through multiprocessing.
- A growing interest in high-end servers as cloud computing and software-as-a-service become more important.
- A growth in data-intensive applications driven by the availability of massive amounts of data on the Internet.
- The insight that increasing performance on the desktop is less important (outside of graphics, at least), either because current performance is acceptable or because highly compute- and data-intensive applications are being done on the cloud.
- An improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant inherent parallelism, arising from large datasets (usually in the form of data parallelism), “natural-world” parallelism (which occurs in scientific and engineering codes), or parallelism among large numbers of independent requests (request-level parallelism).
- The advantages of leveraging a design investment by replication rather than unique design; all multiprocessor designs provide such leverage.

The third quotation reminds us that multicore may provide only limited possibilities for scaling performance. The combination of Amdahl’s Law effects and the end of Dennard scaling mean that the future of multicore may be limited, at least as a method of scaling up the performance of single applications. We return to this topic late in the chapter.

In this chapter, we focus on exploiting thread-level parallelism (TLP). TLP implies the existence of multiple program counters and thus is exploited primarily through MIMDs. Although MIMDs have been around for decades, the movement of thread-level parallelism to the forefront across the range of computing from embedded applications to high-end severs is relatively recent. Likewise, the extensive use of thread-level parallelism for a wide-range of general-purpose applications, versus either transaction processing or scientific applications, is relatively new.

Our focus in this chapter is on *multiprocessors*, which we define as computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space. Such systems exploit thread-level parallelism through two different software models. The first is the execution of a tightly coupled set of threads collaborating on a single task, which is typically called *parallel processing*. The second is the execution of multiple, relatively independent processes that may originate from one or more users, which is a form of *request-level parallelism*, although at a much smaller scale than what we explore in the next chapter. Request-level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries, or multiple applications running independently, often called *multiprogramming*.

The multiprocessors we examine in this chapter typically range in size from a dual processor to dozens and sometimes hundreds of processors and communicate and coordinate through the sharing of memory. Although sharing through memory implies a shared address space, it does not necessarily mean there is a single physical memory. Such multiprocessors include both single-chip systems with multiple cores, known as *multicore*, and computers consisting of multiple chips, each of which is typically a multicore. Many companies make such multiprocessors, including HP, Dell, Cisco, IBM, SGI, Lenovo, Oracle, Fujitsu, and many others.

In addition to true multiprocessors, we will return to the topic of multithreading, a technique that supports multiple threads executing in an interleaved fashion on a single multiple-issue processor. Many multicore processors also include support for multithreading.

In the next chapter, we consider ultrascale computers built from very large numbers of processors, connected with networking technology (not necessarily the same networking technology used to connect computers to the Internet) and often called *clusters*; these large-scale systems are used for cloud computing primarily with massive numbers of independent tasks being executed in parallel. More recently, computationally intensive tasks that can be easily made parallel, such as Search and certain machine learning algorithms have also made use of clusters. When these clusters grow to tens of thousands of servers and beyond, we call them *warehouse-scale computers*. Amazon, Google, Microsoft, and Facebook all make warehouse-scale computers.

In addition to the multiprocessors we study here and the warehouse-scaled systems of the next chapter, there are a range of special large-scale multiprocessor systems, sometimes called *multicomputers*, which are less tightly coupled than the multiprocessors examined in this chapter but usually more tightly coupled than the warehouse-scale systems of the next chapter. The primary use for such multicomputers is in high-end scientific computation, although they are sometimes used for commercial applications filling the niche between multiprocessors and warehouse-scale computers. The Cray X series and IBM BlueGene are typical examples of these multicomputers.

Many other books, such as Culler et al. (1999), cover such systems in detail. Because of the large and changing nature of the field of multiprocessing (the just-mentioned Culler et al. reference is over 1000 pages and discusses only multiprocessing!), we have chosen to focus our attention on what we believe is the most important and general-purpose portions of the computing space. Appendix I discusses some of the issues that arise in building such computers in the context of large-scale scientific applications.

Our focus will be on multiprocessors with roughly 4–256 processor cores, which might occupy anywhere from 4 to 16 separate chips. Such designs vastly dominate in terms of both units and dollars. In large-scale multiprocessors, the interconnection networks are a critical part of the design; Appendix F focuses on that topic.

Multiprocessor Architecture: Issues and Approach

To take advantage of an MIMD multiprocessor with n processors, we must usually have at least n threads or processes to execute; with multithreading, which is present in most multicore chips today, that number is 2–4 times higher. The independent threads within a single process are typically identified by the programmer or created by the operating system (from multiple independent requests). At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the *grain size*, is important in considering how to exploit thread-level parallelism efficiently, the important qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system or programmer and that the threads consist of hundreds to millions of instructions that may be executed in parallel.

Threads can also be used to exploit data-level parallelism, although the overhead is usually higher than would be seen with an SIMD processor or with a GPU (see Chapter 4). This overhead means that grain size must be sufficiently large to exploit the parallelism efficiently. For example, although a vector processor or GPU may be able to efficiently parallelize operations on short vectors, the resulting grain size when the parallelism is split among many threads may be so small that the overhead makes the exploitation of the parallelism prohibitively expensive in an MIMD.

Existing shared-memory multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictates a memory organization and interconnect strategy. We refer to the multiprocessors by their memory organization because what constitutes a small or large number of processors continues to change over time.

The first group, which we call *symmetric (shared-memory) multiprocessors* (SMPs), or *centralized shared-memory multiprocessors*, features small to moderate numbers of cores, typically 32 or fewer. For multiprocessors with such small processor counts, it is possible for the processors to share a single centralized memory that all processors have equal access to, thus the term *symmetric*. In multicore chips, the memory is often shared in a centralized fashion among the cores; most existing multicores are SMPs, but not all. (Note that some literature mistakenly appears to use SMP to stand for Shared Memory Processor, but this usage is erroneous.)

Some multicores have nonuniform access to the outermost cache, a structure called *NUCA* for *Nonuniform Cache Access*, and are thus are not truly SMPs, even if they have a single main memory. The IBM Power8 has distributed L3 caches with nonuniform access time to different addresses in L3.

In multiprocessors consisting of multiple multicore chips, there are often separate memories for each multicore chip. Thus the memory is distributed rather than centralized. As we will see later in the chapter, many designs with distributed memory have fast access to a local memory and much slower access to remote memory; often the differences in access time to various remote memories are small in comparison to the difference between the access times to the local memory and to a remote memory. In such designs, the programmer and software system need to be aware of whether accesses are to local or remote memory, but may be able to ignore the distribution of accesses among remote memories. Because an SMP approach becomes less attractive with a growing number of processors, most of the very largest multiprocessors use some form of distributed memory.

SMP architectures are also sometimes called *uniform memory access* (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organized into multiple banks. Figure 5.1 shows what these multiprocessors look like. The architecture of SMPs is the topic of Section 5.2, and we explain the approach in the context of a multicore.

The alternative design approach consists of multiprocessors with physically distributed memory, called *distributed shared memory* (DSM). Figure 5.2 shows what these multiprocessors look like. To support larger processor counts, memory must be distributed among the processors rather than centralized; otherwise, the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency.

With the rapid increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for

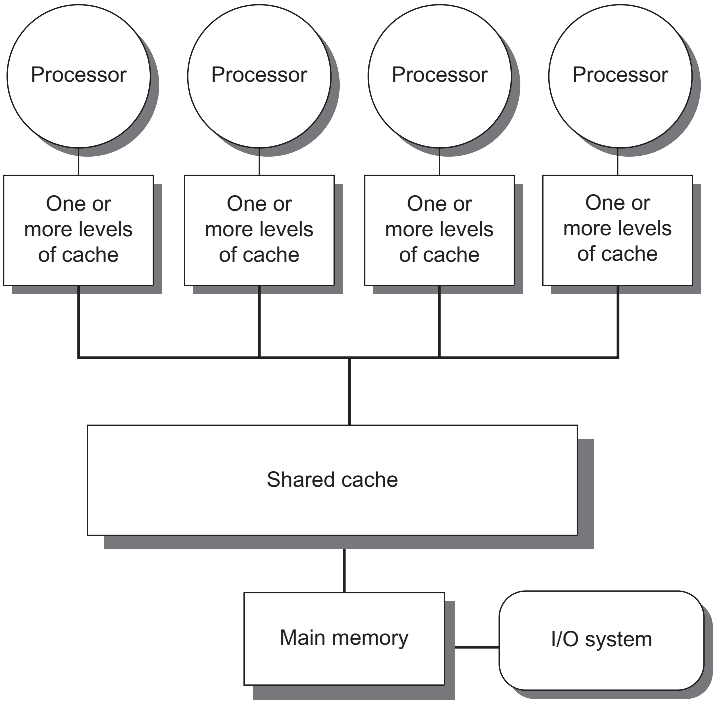

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.

which distributed memory is preferred continues to shrink. The introduction of multicore processors has meant that even some 2-chip multiprocessors, which might have 16–64 processor cores, use distributed memory. The larger number of processors also raises the need for a high-bandwidth interconnect, of which we will see examples in Appendix F. Both directed networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.

Distributing the memory among the nodes both increases the bandwidth and reduces the latency to local memory. A DSM multiprocessor is also called a *NUMA* (nonuniform memory access) because the access time depends on the location of a data word in memory. The key disadvantages for a DSM are that communicating data among processors becomes somewhat more complex and a DSM requires more effort in the software to take advantage of the increased memory bandwidth

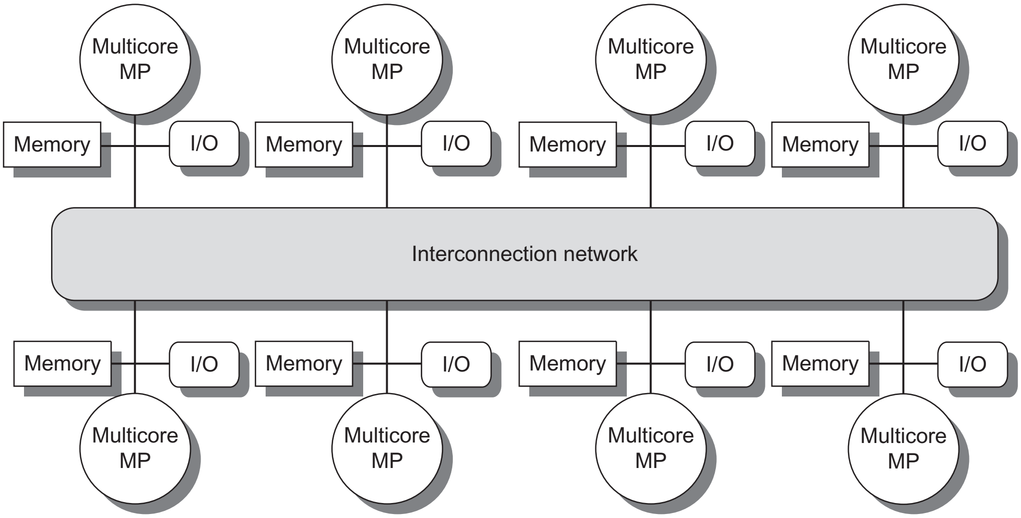


Figure 5.2 The basic architecture of a distributed-memory multiprocessor in 2017 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes. Each processor core shares the entire memory, although the access time to the local memory attached to the core's chip will be much faster than the access time to remote memories.

afforded by distributed memories. Because most multicore-based multiprocessors with more than a few processor chips use distributed memory, we will explain the operation of distributed memory multiprocessors from this viewpoint.

In both SMP and DSM architectures, communication among threads occurs through a shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. The term *shared memory* associated with both SMP and DSM refers to the fact that the *address space* is shared.

In contrast, the clusters and warehouse-scale computers in the next chapter look like individual computers connected by a network, and the memory of one processor cannot be accessed by another processor without the assistance of software protocols running on both processors. In such designs, message-passing protocols are used to communicate data among processors.

Challenges of Parallel Processing

The application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task. Two important hurdles, both explainable with Amdahl's Law, make parallel processing challenging. To overcome these hurdles typically requires a comprehensive approach that addresses the choice of algorithm and its implementation, the underlying programming language and system, the operating system and its support functions, and the architecture and hardware implementation. Although in many instances, one of these is a key bottleneck, when scaling to a larger processor counts (approaching 100 or more), often *all* aspects of the software and hardware need attention.

The first hurdle has to do with the limited parallelism available in programs, and the second arises from the relatively high cost of communications. Limitations in available parallelism make it difficult to achieve good speedups in any parallel processor, as our first example shows.

Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's Law is

$$\text{Speedup} = \frac{1}{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, whereas the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

$$80 = \frac{1}{\dfrac{\text{Fraction}_{\text{parallel}}}{100} + \left(1 - \text{Fraction}_{\text{parallel}}\right)}$$

Simplifying this equation yields:

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times \left(1 - \text{Fraction}_{\text{parallel}}\right) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of the original computation can be sequential! Of course, to achieve linear speedup (speedup of n with n processors), the entire program must usually be parallel with no serial portions. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode. Amdahl's Law can be used to analyze applications with varying amounts of speedup, as the next example shows.

Example Suppose we have an application running on a 100-processor multiprocessor, and assume that application can use 1, 50, or 100 processors. If we assume that 95% of the time we can use all 100 processors, how much of the remaining 5% of the execution time must employ 50 processors if we want a speedup of 80?

Answer We use Amdahl's Law with more terms:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{100}}{\text{Speedup}_{100}} + \frac{\text{Fraction}_{50}}{\text{Speedup}_{50}} + (1 - \text{Fraction}_{100} - \text{Fraction}_{50})}$$

Substituting in:

$$80 = \frac{1}{\frac{0.95}{100} + \frac{\text{Fraction}_{50}}{50} + (1 - 0.95 - \text{Fraction}_{80})}$$

Simplifying:

$$0.76 + 1.6 \times \text{Fraction}_{50} + 4.0 - 80 \times \text{Fraction}_{50} = 1$$
$$4.76 - 78.4 \times \text{Fraction}_{50} = 1$$
$$\text{Fraction}_{50} = 0.048$$

If 95% of an application can use 100 processors perfectly, to get a speedup of 80, 4.8% of the remaining time must be spent using 50 processors and only 0.2% can be serial!

The second major challenge in parallel processing involves the large latency of remote access in a parallel processor. In existing shared-memory multiprocessors, communication of data between separate cores may cost 35–50 clock cycles and among cores on separate chips anywhere from 100 clock cycles to as much as 300 or more clock cycles (for large-scale multiprocessors), depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor. The effect of long communication delays is clearly substantial. Let's consider a simple example.

Example Suppose we have an application running on a 32-processor multiprocessor that has a 100 ns delay to handle a reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is obviously optimistic. Processors are stalled on a remote request, and the processor clock rate is 4 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Answer It is simpler to first calculate the clock cycles per instruction. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost} \\ &= 0.5 + 0.2\% \times \text{Remote request cost}\end{aligned}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{100\,\text{ns}}{0.25\,\text{ns}} = 400\,\text{cycles}$$

Therefore we can compute the CPI:

$$\begin{aligned}\text{CPI} &= 0.5 + 0.20\% \times 400 \\ &= 1.3\end{aligned}$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex because some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be worse because contention caused by many references trying to use the global interconnect can lead to increased delays, or the access time might be better if memory were distributed and the access was to the local memory.

This problem could have also been analyzed using Amdahl's Law, an exercise we leave to the reader.

These problems—insufficient parallelism and long-latency remote communication—are the two biggest performance challenges in using multiprocessors. The problem of inadequate application parallelism must be attacked primarily in software with new algorithms that offer better parallel performance, as well as by software systems that maximize the amount of time spent executing with the full complement of processors. Reducing the impact of long remote latency can be attacked both by the architecture and by the programmer. For example, we can reduce the frequency of remote accesses with either hardware mechanisms, such as caching shared data, or software mechanisms, such as restructuring the data to make more accesses local. We can try to tolerate the latency by using multithreading (discussed later in this chapter) or by using prefetching (a topic we cover extensively in Chapter 2).

Much of this chapter focuses on techniques for reducing the impact of long remote communication latency. For example, Sections 5.2 through 5.4 discuss how caching can be used to reduce remote access frequency, while maintaining a coherent view of memory. Section 5.5 discusses synchronization, which, because it inherently involves interprocessor communication and also can limit parallelism, is a major potential bottleneck. Section 5.6 covers latency-hiding techniques and memory consistency models for shared memory. In Appendix I, we focus primarily on larger-scale multiprocessors that are used predominantly for scientific work.

In that appendix, we examine the nature of such applications and the challenges of achieving speedup with dozens to hundreds of processors.

5.2 Centralized Shared-Memory Architectures

The observation that the use of large, multilevel caches can substantially reduce the memory bandwidth demands of a processor is the key insight that motivates centralized memory multiprocessors. Originally, these processors were all single-core and often took an entire board, and memory was located on a shared bus. With more recent, higher-performance processors, the memory demands have outstripped the capability of reasonable buses, and recent microprocessors directly connect memory to a single chip, which is sometimes called a *backside* or *memory bus* to distinguish it from the bus used to connect to I/O. Accessing a chip's local memory whether for an I/O operation or for an access from another chip requires going through the chip that "owns" that memory. Thus access to memory is asymmetric: faster to the local memory and slower to the remote memory. In a multicore that memory is shared among all the cores on a single chip, but the asymmetric access to the memory of one multicore from the memory of another usually remains.

Symmetric shared-memory machines usually support the caching of both shared and private data. *Private data* are used by a single processor, while *shared data* are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data. When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Because no other processor uses the data, the program behavior is identical to that in a uniprocessor. When shared data are cached, the shared value may be replicated in multiple caches. In addition to the reduction in access latency and required memory bandwidth, this replication also provides a reduction in contention that may exist for shared data items that are being read by multiple processors simultaneously. Caching of shared data, however, introduces a new problem: cache coherence.

What Is Multiprocessor Cache Coherence?

Unfortunately, caching shared data introduces a new problem. Because the view of memory held by two different processors is through their individual caches, the processors could end up seeing different values for the same memory location, as Figure 5.3 illustrates. This difficulty is generally referred to as the *cache coherence problem*. Notice that the coherence problem exists because we have both a global state, defined primarily by the main memory, and a local state, defined by the individual caches, which are private to each processor core. Thus, in a multicore where some level of caching may be shared (e.g., an L3), although some levels are private (e.g., L1 and L2), the coherence problem still exists and must be solved.

Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item. This definition,

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Figure 5.3 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!

although intuitively appealing, is vague and simplistic; the reality is much more complex. This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs. The first aspect, called *coherence*, defines what values can be returned by a read. The second aspect, called *consistency*, determines when a written value will be returned by a read. Let's look at coherence first.

A memory system is coherent if

1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.
2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.
3. Writes to the same location are *serialized*; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

The first property simply preserves program order—we expect this property to be true even in uniprocessors. The second property defines the notion of what it means to have a coherent view of memory: if a processor could continuously read an old data value, we would clearly say that memory was incoherent.

The need for write serialization is more subtle, but equally important. Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X. Serializing the writes ensures that every processor will see the write done by P2 at some point. If we did not serialize the writes, it might be the case that some processors could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely. The simplest way to avoid such difficulties is to ensure that all writes to the same location are seen in the same order; this property is called *write serialization*.

Although the three properties just described are sufficient to ensure coherence, the question of when a written value will be seen is also important. To see why, observe that we cannot require that a read of X instantaneously see the value written for X by some other processor. If, for example, a write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point. The issue of exactly *when* a written value must be seen by a reader is defined by a *memory consistency model*—a topic discussed in Section 5.6.

Coherence and consistency are complementary: *Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations*. For now, make the following two assumptions. First, a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write. Second, the processor does not change the order of any write with respect to any other memory access. These two conditions mean that, if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A. These restrictions allow the processor to reorder reads, but forces the processor to finish a write in program order. We will rely on this assumption until we reach Section 5.6, where we will see exactly the implications of this definition, as well as the alternatives.

Basic Schemes for Enforcing Coherence

The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution. Unlike I/O, where multiple data copies are a rare event—one to be avoided whenever possible—a program running on multiple processors will normally have copies of the same data in several caches. In a coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items.

Coherent caches provide migration because a data item can be moved to a local cache and used there in a transparent fashion. This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.

Because the caches make a copy of the data item in the local cache, coherent caches also provide replication for shared data that are being read simultaneously. Replication reduces both latency of access and contention for a read shared data item. Supporting this migration and replication is critical to performance in accessing shared data. Thus, rather than trying to solve the problem by avoiding it in software, multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

The protocols to maintain coherence for multiple processors are called *cache coherence protocols*. Key to implementing a cache coherence protocol is tracking the state of any sharing of a data block. The state of any cache block is kept using status bits associated with the block, similar to the valid and dirty bits kept in a uniprocessor cache. There are two classes of protocols in use, each of which uses different techniques to track the sharing status:

- *Directory based*—The sharing status of a particular block of physical memory is kept in one location, called the *directory*. There are two very different types of directory-based cache coherence. In an SMP, we can use one centralized directory, associated with the memory or some other single serialization point, such as the outermost cache in a multicore. In a DSM, it makes no sense to have a single directory because that would create a single point of contention and make it difficult to scale to many multicore chips given the memory demands of multicores with eight or more cores. Distributed directories are more complex than a single directory, and such designs are the subject of Section 5.4.
- *Snooping*—Rather than keeping the state of sharing in a single directory, every cache that has a copy of the data from a block of physical memory could track the sharing status of the block. In an SMP, the caches are typically all accessible via some broadcast medium (e.g., a bus connects the per-core caches to the shared cache or memory), and all cache controllers monitor or *snoop* on the medium to determine whether they have a copy of a block that is requested on a bus or switch access. Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore.

Snooping protocols became popular with multiprocessors using microprocessors (single-core) and caches attached to a single shared memory by a bus. The bus provided a convenient broadcast medium to implement the snooping protocols. Multicore architectures changed the picture significantly because all multicores share some level of cache on the chip. Thus some designs switched to using directory protocols, since the overhead was small. To allow the reader to become familiar with both types of protocols, we focus on a snooping protocol here and discuss a directory protocol when we come to DSM architectures.

Snooping Coherence Protocols

There are two ways to maintain the coherence requirement described in the prior section. One method is to ensure that a processor has exclusive access to a data item before writing that item. This style of protocol is called a *write invalidate protocol* because it invalidates other copies on a write. It is by far the most common protocol. Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated.

Figure 5.4 shows an example of an invalidation protocol with write-back caches in action. To see how this protocol ensures coherence, consider a write followed by a read by another processor: because the write requires exclusive

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Figure 5.4 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, processor A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional status bit indicating ownership of a block. The ownership bit indicates that a block may be shared for reads, but only the owning processor can write the block, and that processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 caches for each core. It is this observation that led some designers to opt for a directory protocol within the multicore. To make this work, the L3 cache must be inclusive; recall from Chapter 2, that a cache is inclusive if any location in a higher level cache (L1 and L2 in this case) is also in L3. We return to the topic of inclusion on page 423.

access, any copy held by the reading processor must be invalidated (thus the protocol name). Therefore when the read occurs, it misses in the cache and is forced to fetch a new copy of the data. For a write, we require that the writing processor has exclusive access, preventing any other processor from being able to write simultaneously. If two processors do attempt to write the same data simultaneously, one of them wins the race (we'll see how we decide who wins shortly), causing the other processor's copy to be invalidated. For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. Therefore this protocol enforces write serialization.

The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a *write update* or *write broadcast* protocol. Because a write update protocol must broadcast all writes to shared cache lines, it consumes considerably more bandwidth. For this reason, virtually all recent multiprocessors have opted to implement a write invalidate protocol, and we will focus only on invalidate protocols for the rest of the chapter.

Basic Implementation Techniques

The key to implementing an invalidate protocol in a multicore is the use of the bus, or another broadcast medium, to perform invalidates. In older multiple-chip multiprocessors, the bus used for coherence is the shared-memory access bus. In a single-chip multicore, the bus can be the connection between the private caches (L1 and L2 in the Intel i7) and the shared outer cache (L3 in the i7). To perform an invalidate, the processor simply acquires bus access and broadcasts the address to be invalidated on the bus. All processors continuously snoop on the bus, watching the addresses. The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache are invalidated.

When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation. If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus. The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated. If the processors were attempting to write the same block, the serialization enforced by the bus would also serialize their writes. One implication of this scheme is that a write to a shared data item cannot actually complete until it obtains bus access. All coherence schemes require some method of serializing accesses to the same cache block, either by serializing access to the communication medium or to another shared structure.

In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a cache miss occurs. In a write-through cache, it is easy to find the recent value of a data item because all written data are always sent to the memory, from which the most recent value of a data item can always be fetched. (Write buffers can lead to some additional complexities and must effectively be treated as additional cache entries.)

For a write-back cache, the problem of finding the most recent data value is harder because the most recent value of a data item can be in a private cache rather than in the shared cache or memory. Fortunately, write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops every address placed on the shared bus. If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory (or L3) access to be aborted. The additional complexity comes from having to retrieve the cache block from another processor's private cache (L1 or L2), which can often take longer than retrieving it from L3. Because write-back caches generate lower requirements for memory bandwidth, they can support larger numbers of faster processors. As a result, all multicore processors use write-back at the outermost levels of the cache, and we will examine the implementation of coherence with write-back caches.

The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement. Read misses, whether generated by an invalidation or by some other event, are also straightforward because they simply rely on the snooping capability. For writes, we want to know whether any other copies of the block are cached because, if there are no other cached copies, then the write does not need to be placed on the bus in a write-back cache. Not sending the write reduces both the time to write and the required bandwidth.

To track whether or not a cache block is shared, we can add an extra state bit associated with each cache block, just as we have a valid bit and a dirty bit. By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate. When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as *exclusive*. No further invalidations will be sent by that core for that block. The core with the sole copy of a cache block is normally called the *owner* of the cache block.

When an invalidation is sent, the state of the owner's cache block is changed from shared to unshared (or exclusive). If another processor later requests this cache block, the state must be made shared again. Because our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared.

Every bus transaction must check the cache-address tags, which could potentially interfere with processor cache accesses. One way to reduce this interference is to duplicate the tags and have snoop accesses directed to the duplicate tags. Another approach is to use a directory at the shared L3 cache; the directory indicates whether a given block is shared and possibly which cores have copies. With the directory information, invalidates can be directed only to those caches with copies of the cache block. This requires that L3 must always have a copy of any data item in L1 or L2, a property called *inclusion*, which we will return to in Section 5.7.

An Example Protocol

A snooping coherence protocol is usually implemented by incorporating a finite-state controller in each core. This controller responds to requests from the processor in the core and from the bus (or other broadcast medium), changing the state of the selected cache block, as well as using the bus to access data or to invalidate it. Logically, you can think of a separate controller as being associated with each block; that is, snooping operations or cache requests for different blocks can proceed independently. In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion (i.e., one operation may be initiated before another is completed, even though only one cache access or one bus access is allowed at a time). Also, remember that, although we refer to a bus in the following description, any interconnection network that supports a broadcast to all the coherence controllers and their associated private caches can be used to implement snooping.

The simple protocol we consider has three states: invalid, shared, and modified. The shared state indicates that the block in the private cache is potentially shared, whereas the modified state indicates that the block has been updated in the private cache; note that the modified state *implies* that the block is exclusive. Figure 5.5 shows the requests generated by a core (in the top half of the table) as well as those coming from the bus (in the bottom half of the table). This protocol is for a write-back cache but is easily changed to work for a write-through cache by reinterpreting the modified state as an exclusive state and updating the cache on writes in the normal fashion for a write-through cache. The most common extension of this basic protocol is the addition of an exclusive state, which describes a block that is unmodified but held in only one private cache. We describe this and other extensions on page 388.

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Figure 5.5 The cache coherence mechanism receives requests from both the core's processor and the shared bus and responds to these based on the type of request, whether it hits or misses in the local cache, and the state of the local cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read, misses, write misses, or invalidates snooped from the bus, an action is required *only* if the read or write addresses match a block in the local cache and the block is valid.

When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the cache block invalidate it. For a write miss in a write-back cache, if the block is exclusive in just one private cache, that cache also writes back the block; otherwise, the data can be read from the shared cache or memory.

Figure 5.6 shows a finite-state transition diagram for a single private cache block using a write invalidation protocol and a write-back cache. For simplicity, the three states of the protocol are duplicated to represent transitions based on

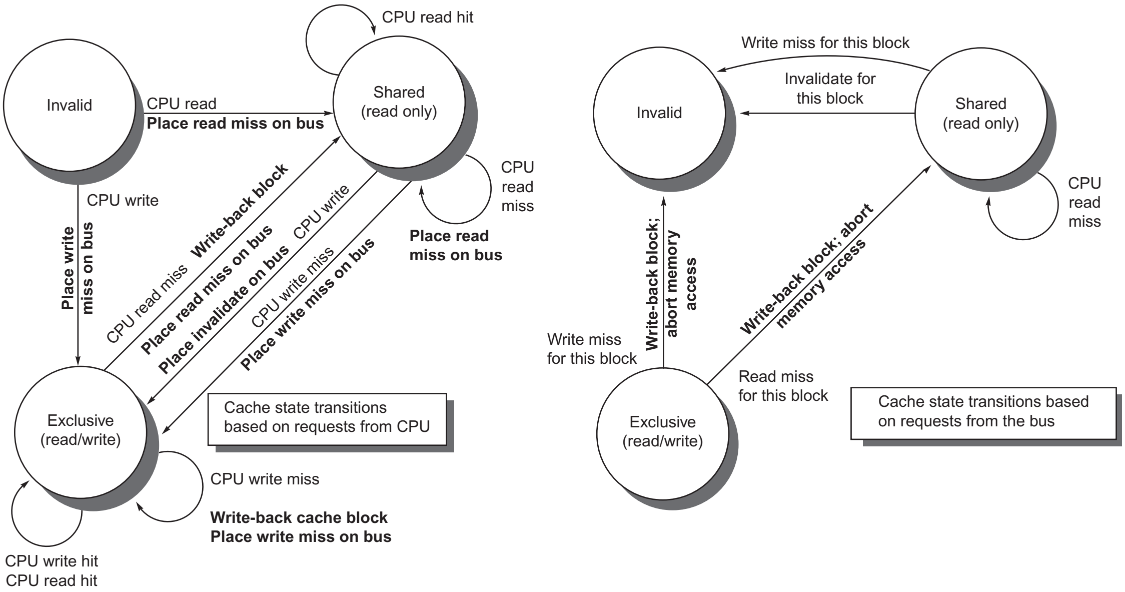

CPU read hit
Invalid
CPU read
Place read miss on bus
Shared
(read only)
CPU read miss
Place read miss on bus
CPU write
Place write miss on bus
CPU read miss
Write-back block
Place read miss on bus
CPU write
Place invalidate on bus
CPU write miss
Place write miss on bus
Exclusive
(read/write)
Cache state transitions based on requests from CPU
CPU write miss
Write-back cache block
Place write miss on bus
CPU write hit
CPU read hit
Write miss for this block
Invalidate for this block
Invalid
Shared
(read only)
CPU read miss
Write-back block; abort memory access
Write-back block; abort memory access
Write miss for this block
Read miss for this block
Exclusive
(read/write)
Cache state transitions based on requests from the bus

Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache. The cache states are shown in circles, with any access permitted by the local processor without a state transition shown in parentheses under the name of the state. The stimulus causing a state change is shown on the transition arcs in regular type, and any bus actions generated as part of the state transition are shown on the transition arc in *bold*. The stimulus actions apply to a block in the private cache, not to a specific address in the cache. Thus a read miss to a block in the shared state is a miss for that cache block but for a different address. The left side of the diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus. A read miss in the exclusive or shared state and a write miss in the exclusive state occur when the address requested by the processor does not match the address in the local cache block. Such a miss is a standard cache replacement miss. An attempt to write a block in the shared state generates an invalidate. Whenever a bus transaction occurs, all private caches that contain the cache block specified in the bus transaction take the action dictated by the right half of the diagram. The protocol assumes that memory (or a shared cache) provides data on a read miss for a block that is clean in all local caches. In actual implementations, these two sets of state diagrams are combined. In practice, there are many subtle variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss. In a multicore chip, the shared cache (usually L3, but sometimes L2) acts as the equivalent of memory, and the bus is the bus between the private caches of each core and the shared cache, which in turn interfaces to the memory.

processor requests (on the left, which corresponds to the top half of the table in Figure 5.5), as opposed to transitions based on bus requests (on the right, which corresponds to the bottom half of the table in Figure 5.5). Boldface type is used to distinguish the bus actions, as opposed to the conditions on which a state transition depends. The state in each node represents the state of the selected private cache block specified by the processor or bus request.

All of the states in this cache protocol would be needed in a uniprocessor cache, where they would correspond to the invalid, valid (and clean), and dirty states. Most of the state changes indicated by arcs in the left half of Figure 5.6 would be needed in a write-back uniprocessor cache, with the exception being the invalidate on a write hit to a shared block. The state changes represented by the arcs in the right half of Figure 5.6 are needed only for coherence and would not appear at all in a uniprocessor cache controller.

As mentioned earlier, there is only one finite-state machine per cache, with stimuli coming either from the attached processor or from the bus. Figure 5.7 shows how the state transitions in the right half of Figure 5.6 are combined with those in the left half of the figure to form a single state diagram for each cache block.

To understand why this protocol works, observe that any valid cache block is either in the shared state in one or more private caches or in the exclusive state in exactly one cache. Any transition to the exclusive state (which is required for a processor to write to the block) requires an invalidate or write miss to be placed on the bus, causing all local caches to make the block invalid. In addition, if some other local cache had the block in exclusive state, that local cache generates a write-back, which supplies the block containing the desired address. Finally, if a read miss occurs on the bus to a block in the exclusive state, the local cache with the exclusive copy changes its state to shared.

The actions in gray in Figure 5.7, which handle read and write misses on the bus, are essentially the snooping component of the protocol. One other property that is preserved in this protocol, and in most other protocols, is that any memory block in the shared state is always up to date in the outer shared cache (L2 or L3, or memory if there is no shared cache), which simplifies the implementation. In fact, it does not matter whether the level out from the private caches is a shared cache or memory; the key is that all accesses from the cores go through that level.

Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier. The most important of these is that the protocol assumes that operations are *atomic*—that is, an operation can be done in such a way that no intervening operation can occur. For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action. In reality this is not true. In fact, even a read miss might not be atomic; after detecting a miss in the L2 of a multicore, the core must arbitrate for access to the bus connecting to the shared L3. Nonatomic actions introduce the possibility that the protocol can *deadlock*, meaning that it reaches a state where it cannot continue. We will explore these complications later in this section and when we examine DSM designs.

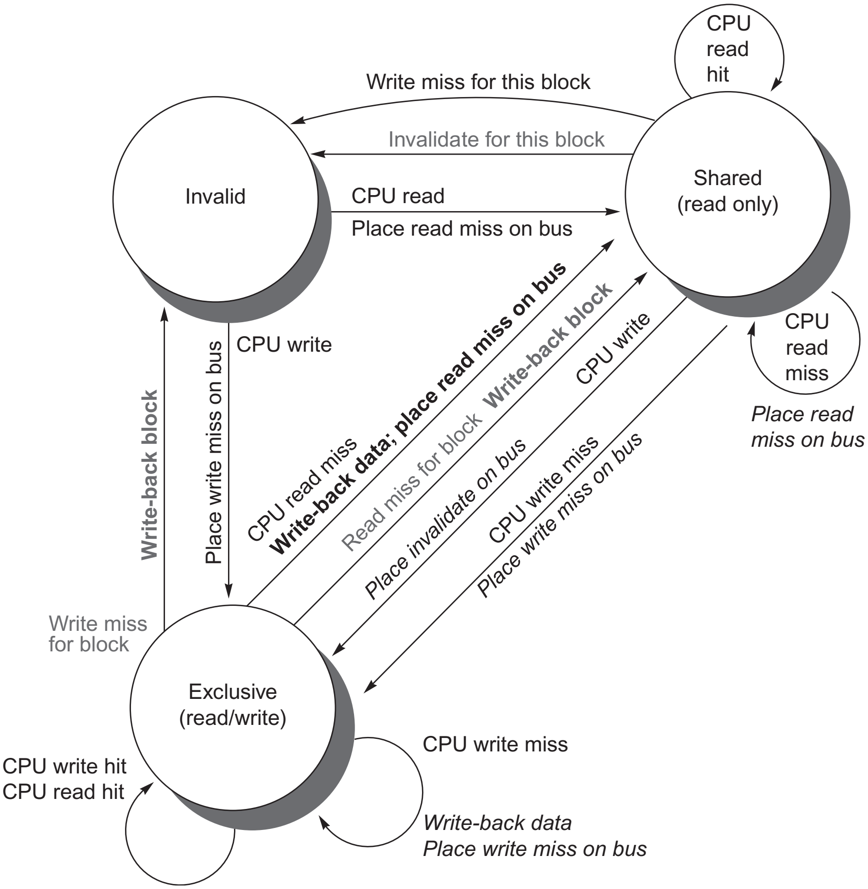

CPU read hit
Write miss for this block
Invalidate for this block
Invalid
Shared (read only)
CPU read
Place read miss on bus
CPU write
Place write miss on bus
Write-back block
CPU read miss
Write-back data; place read miss on bus
Read miss for block
Write-back block
CPU write
Place invalidate on bus
CPU write miss
Place write miss on bus
CPU read miss
Place read miss on bus
Write miss for block
Exclusive (read/write)
CPU write hit
CPU read hit
CPU write miss
Write-back data
Place write miss on bus

Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in *black* and by the bus activities shown in *gray*. As in Figure 5.6, the activities on a transition are shown in *bold*.

With multicore processors, the coherence among the processor cores is all implemented on chip, using either a snooping or simple central directory protocol. Many multiprocessor chips, including the Intel Xeon and AMD Opteron, support multichip multiprocessors that could be built by connecting a high-speed interface already incorporated in the chip. These next-level interconnects are not just extensions of the shared bus, but use a different approach for interconnecting multicores.

A multiprocessor built with multiple multicore chips will usually have a distributed memory architecture and will need an interchip coherency mechanism above and beyond the one within the chip. In most cases, some form of directory scheme is used.

Extensions to the Basic Coherence Protocol

The coherence protocol we have just described is a simple three-state protocol and is often referred to by the first letter of the states, making it a MSI (Modified, Shared, Invalid) protocol. There are many extensions of this basic protocol, which we mention in the captions of figures in this section. These extensions are created by adding additional states and transactions that optimize certain behaviors, possibly resulting in improved performance. Two of the most common extensions are

1. *MESI* adds the state Exclusive to the basic MSI protocol, yielding four states (Modified, Exclusive, Shared, and Invalid). The exclusive state indicates that a cache block is resident in only a single cache but is clean. If a block is in the E state, it can be written without generating any invalidates, which optimizes the case where a block is read by a single cache before being written by that same cache. Of course, when a read miss to a block in the E state occurs, the block must be changed to the S state to maintain coherence. Because all subsequent accesses are snooped, it is possible to maintain the accuracy of this state. In particular, if another processor issues a read miss, the state is changed from exclusive to shared. The advantage of adding this state is that a subsequent write to a block in the exclusive state by the same core need not acquire bus access or generate an invalidate, since the block is known to be exclusively in this local cache; the processor merely changes the state to modified. This state is easily added by using the bit that encodes the coherent state as an exclusive state and using the dirty bit to indicate that a bock is modified. The Intel i7 uses a variant of a MESI protocol, called MESIF, which adds a state (Forward) to designate which sharing processor should respond to a request. It is designed to enhance performance in distributed memory organizations.
2. *MOESI* adds the state Owned to the MESI protocol to indicate that the associated block is owned by that cache and out-of-date in memory. In MSI and MESI protocols, when there is an attempt to share a block in the Modified state, the state is changed to Shared (in both the original and newly sharing cache), and the block must be written back to memory. In a MOESI protocol, the block can be changed from the Modified to Owned state in the original cache without writing it to memory. Other caches, which are newly sharing the block, keep the block in the Shared state; the O state, which only the original cache holds, indicates that the main memory copy is out of date and that the designated cache is the owner. The owner of the block must supply it on a miss, since memory is not up to date and must write the block back to memory if it is replaced. The AMD Opteron processor family uses the MOESI protocol.

The next section examines the performance of these protocols for our parallel and multiprogrammed workloads; the value of these extensions to a basic protocol will be clear when we examine the performance. But, before we do that, let’s take a brief look at the limitations on the use of a symmetric memory structure and a snooping coherence scheme.

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

As the number of processors in a multiprocessor grows, or as the memory demands of each processor grow, any centralized resource in the system can become a bottleneck. For multicores, a single shared bus became a bottleneck with only a few cores. As a result, multicore designs have gone to higher bandwidth interconnection schemes, as well as multiple, independent memories to allow larger numbers of cores. The multicore chips we examine in Section 5.8 use three different approaches:

1. The IBM Power8, which has up to 12 processors in a single multicore, uses 8 parallel buses that connect the distributed L3 caches and up to 8 separate memory channels.
2. The Xeon E7 uses three rings to connect up to 32 processors, a distributed L3 cache, and two or four memory channels (depending on the configuration).
3. The Fujitsu SPARC64 X+ uses a crossbar to connect a shared L2 to up to 16 cores and multiple memory channels.

The SPARC64 X+ is a symmetric organization with uniform access time. The Power8 has nonuniform access time for both L3 and memory. Although the uncontended access time differences among memory addresses within a single Power8 multicore are not large, with contention for memory, the access time differences can become significant even within one chip. The Xeon E7 can operate as if access times were uniform; in practice, software systems usually organize memory so that the memory channels are associated with a subset of the cores.

Snooping bandwidth at the caches can also become a problem because every cache must examine every miss, and having additional interconnection bandwidth only pushes the problem to the cache. To understand this problem, consider the following example.

Example Consider an 8-processor multicore where each processor has its own L1 and L2 caches, and snooping is performed on a shared bus among the L2 caches. Assume the average L2 request, whether for a coherence miss or other miss, is 15 cycles. Assume a clock rate of 3.0 GHz, a CPI of 0.7, and a load/store frequency of 40%. If our goal is that no more than 50% of the L2 bandwidth is consumed by coherence traffic, what is the maximum coherence miss rate per processor?

Answer Start with an equation for the number of cache cycles that can be used (where CMR is the coherence miss rate):

$$\text{Cache cycles available} = \frac{\text{Clock rate}}{\text{Cycles per request} \times 2} = \frac{3.0\text{Ghz}}{30} = 0.1 \times 10^9$$

$$\begin{aligned}\text{Cache cycles available} &= \text{Memory references/clock/processor} \times \text{Clock rate}\\ &\quad \times \text{processor count} \times \text{CMR}\\ &= \frac{0.4}{0.7} \times 3.0\text{GHz} \times 8 \times \text{CMR} = 13.7 \times 10^9 \times \text{CMR}\end{aligned}$$

$$\text{CMR} = \frac{0.1}{13.7} = 0.0073 = 0.73\%$$

This means that the coherence miss rate must be 0.73% or less. In the next section, we will see several applications with coherence miss rates in excess of 1%. Alternatively, if we assume that CMR can be 1%, then we could support just under 6 processors. Clearly, even small multicores will require a method for scaling snoop bandwidth.

There are several techniques for increasing the snoop bandwidth:

1. As mentioned earlier, the tags can be duplicated. This doubles the effective cache-level snoop bandwidth. If we assume that half the coherence requests do not hit on a snoop request and the cost of the snoop request is only 10 cycles (versus 15), then we can cut the average cost of a CMR to 12.5 cycles. This reduction allows the coherence miss rate to be 0.88, or alternatively to support one additional processor (7 versus 6).
2. If the outermost cache on a multicore (typically L3) is shared, we can distribute that cache so that each processor has a portion of the memory and handles snoops for that portion of the address space. This approach, used by the IBM 12-core Power8, leads to a NUCA design, but effectively scales the snoop bandwidth at L3 by the number of processors. If there is a snoop hit in L3, then we must still broadcast to all L2 caches, which must in turn snoop their contents. Since L3 is acting as a filter on the snoop requests, L3 must be inclusive.
3. We can place a directory at the level of the outermost shared cache (say, L3). L3 acts as a filter on snoop requests and must be inclusive. The use of a directory at L3 means that we need not snoop or broadcast to all the L2 caches, but only those that the directory indicates may have a copy of the block. Just as L3 may be distributed, the associated directory entries may also be distributed. This approach is used in the Intel Xeon E7 series, which supports from 8 to 32 cores.

Figure 5.8 shows how a multicore with a distributed cache system, such as that used in schemes 2 or 3, might look. If additional multicore chips were added to form a larger multiprocessor, an off-chip network would needed, as well as a method to extend the coherence mechanisms (as we will see in Section 5.8).

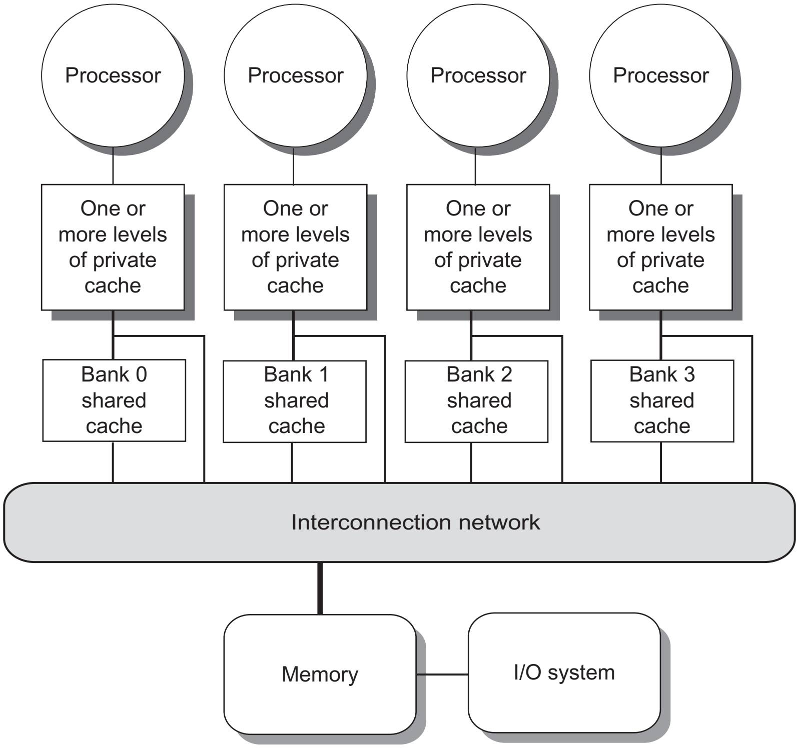

Processor
Processor
Processor
Processor
One or more levels of private cache
One or more levels of private cache
One or more levels of private cache
One or more levels of private cache
Bank 0 shared cache
Bank 1 shared cache
Bank 2 shared cache
Bank 3 shared cache
Interconnection network
Memory
I/O system

Figure 5.8 A single-chip multicore with a distributed cache. In current designs, the distributed shared cache is usually L3, and levels L1 and L2 are private. There are typically multiple memory channels (2–8 in today's designs). This design is NUCA, since the access time to L3 portions varies with faster access time for the directly attached core. Because it is NUCA, it is also NUMA.

The AMD Opteron represents another intermediate point in the spectrum between a snooping and a directory protocol. Memory is directly connected to each multicore chip, and up to four multicore chips can be connected. The system is a NUMA because local memory is somewhat faster. The Opteron implements its coherence protocol using the point-to-point links to broadcast up to three other chips. Because the interprocessor links are not shared, the only way a processor can know when an invalid operation has completed is by an explicit acknowledgment. Thus the coherence protocol uses a broadcast to find potentially shared copies, like a snooping protocol, but uses the acknowledgments to order operations, like a directory protocol. Because local memory is only somewhat faster than remote memory in the Opteron implementation, some software treats the Opteron multiprocessor as having uniform memory access.

In Section 5.4, we examine directory-based protocols, which eliminate the need for broadcast to all caches on a miss. Some multicore designs use directories within the multicore (Intel Xeon E7), while others add directories when scaling beyond a multicore. Distributed directories eliminate the need for a single point to serialize all accesses (typically a single shared bus in a snooping scheme), and any scheme that removes the single point of serialization must deal with many of the same challenges as a distributed directory scheme.

Implementing Snooping Cache Coherence

The devil is in the details.

Classic proverb

When we wrote the first edition of this book in 1990, our final "Putting It All Together" was a 30-processor, single-bus multiprocessor using snoop-based coherence; the bus had a capacity of just over 50 MiB/s, which would not be enough bus bandwidth to support even one core of an Intel i7 in 2017! When we wrote the second edition of this book in 1995, the first cache coherence multiprocessors with more than a single bus had recently appeared, and we added an appendix describing the implementation of snooping in a system with multiple buses. In 2017, *every* multicore multiprocessor system that supports 8 or more cores uses an interconnect other than a single bus, and designers must face the challenge of implementing snooping (or a directory scheme) without the simplification of a bus to serialize events.

As we observed on page 386, the major complication in actually implementing the snooping coherence protocol we have described is that write and upgrade misses are not atomic in any recent multiprocessor. The steps of detecting a write or upgrade miss, communicating with the other processors and memory, getting the most recent value for a write miss and ensuring that any invalidates are processed, and updating the cache cannot be done as though they took a single cycle.

In a multicore with a single bus, these steps can be made effectively atomic by arbitrating for the bus to the shared cache or memory first (before changing the cache state) and not releasing the bus until all actions are complete. How can the processor know when all the invalidates are complete? In early designs, a single line was used to signal when all necessary invalidates had been received and were being processed. Following that signal, the processor that generated the miss could release the bus, knowing that any required actions would be completed before any activity related to the next miss. By holding the bus exclusively during these steps, the processor effectively made the individual steps atomic.

In a system without a single, central bus, we must find some other method of making the steps in a miss atomic. In particular, we must ensure that two processors that attempt to write the same block at the same time, a situation which is called a *race*, are strictly ordered: one write is processed and precedes before the next is begun. It does not matter which of two writes in a race wins the race, just that there be only a single winner whose coherence actions are completed first. In a multicore using multiple buses, races can be eliminated if each block of memory is associated with only a single bus, ensuring that two attempts to access the same block must be serialized by that common bus. This property, together with the ability to restart the miss handling of the loser in a race, are the keys to implementing snooping cache coherence without a bus. We explain the details in Appendix I.

It is possible to combine snooping and directories, and several designs use snooping within a multicore and directories among multiple chips or a combination of directories at one cache level and snooping at another level.

5.3 Performance of Symmetric Shared-Memory Multiprocessors

In a multicore using a snooping coherence protocol, several different phenomena combine to determine performance. In particular, the overall cache performance is a combination of the behavior of uniprocessor cache miss traffic and the traffic caused by communication, which results in invalidations and subsequent cache misses. Changing t he processor count, cache size, and block size can affect these two components of the miss rate in different ways, leading to overall system behavior that is a combination of the two effects.

Appendix B breaks the uniprocessor miss rate into the three C's classification (capacity, compulsory, and conflict) and provides insight into both application behavior and potential improvements to the cache design. Similarly, the misses that arise from interprocessor communication, which are often called *coherence misses*, can be broken into two separate sources.

The first source is the *true sharing misses* that arise from the communication of data through the cache coherence mechanism. In an invalidation-based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block. Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred. Both these misses are classified as true sharing misses because they directly arise from the sharing of data among processors.

The second effect, called *false sharing*, arises from the use of an invalidation-based coherence algorithm with a single valid bit per cache block. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. If the word written into is actually used by the processor that received the invalidate, then the reference was a true sharing reference and would have caused a miss independent of the block size. If, however, the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a false sharing miss. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word. The following example makes the sharing patterns clear.

Example Assume that words z1 and z2 are in the same cache block, which is in the shared state in the caches of both P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.

Time	P1	P2
1	Write z1	
2		Read z2
3	Write z1	
4		Write z2
5	Read z2	

Answer Here are the classifications by time step:

1. This event is a true sharing miss, since z1 is in the shared state in P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since z2 was invalidated by the write of z1 in P1, but that value of z1 is not used in P2.
3. This event is a false sharing miss, since the block containing z1 is marked shared due to the read in P2, but P2 did not read z1. The cache block containing z1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols, this will be handled as an *upgrade request*, which generates a bus invalidate, but does not transfer the cache block.
4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2.

Although we will see the effects of true and false sharing misses in commercial workloads, the role of coherence misses is more significant for tightly coupled applications that share significant amounts of user data. We examine their effects in detail in Appendix I when we consider the performance of a parallel scientific workload.

A Commercial Workload

In this section, we examine the memory system behavior of a 4-processor shared-memory multiprocessor when running an online transaction processing workload. The study we examine was done with a 4-processor Alpha system in 1998, but it remains the most comprehensive and insightful study of the performance of a multiprocessor for such workloads. We will focus on understanding the multiprocessor cache activity, and particularly the behavior in L3, where much of the traffic is coherence-related.

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct-mapped	8-way I/8-way D
	Block size	32 B	64 B
	Miss penalty	7	10
L2	Size	96 KB	256 KB
	Associativity	3-way	8-way
	Block size	32 B	64 B
	Miss penalty	21	35
L3	Size	2 MiB (total 8 MiB unshared)	2 MiB per core (8 MiB total shared)
	Associativity	Direct-mapped	16-way
	Block size	64 B	64 B
	Miss penalty	80	~100

Figure 5.9 The characteristics of the cache hierarchy of the Alpha 21164 used in this study and the Intel i7. Although the sizes are larger and the associativity is higher on the i7, the miss penalties are also higher, so the behavior may differ only slightly. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. A key difference is that L3 is shared in the i7 versus four separate, unshared caches in the Alpha server.

The results were collected either on an AlphaServer 4100 or using a configurable simulator modeled after the AlphaServer 4100. Each processor in the AlphaServer 4100 is an Alpha 21164, which issues up to four instructions per clock and runs at 300 MHz. Although the clock rate of the Alpha processor in this system is considerably slower than processors in systems designed in 2017, the basic structure of the system, consisting of a four-issue processor and a three-level cache hierarchy, is very similar to the multicore Intel i7 and other processors, as shown in Figure 5.9. Rather than focus on the performance details, we consider data that looks at the simulated L3 behavior for L3 caches varying from 2 to 8 MiB per processor.

Although the original study considered three different workloads, we focus our attention on the online transaction-processing (OLTP) workload modeled after TPC-B (which has memory behavior similar to its newer cousin TPC-C, described in Chapter 1) and using Oracle 7.3.2 as the underlying database. The workload consists of a set of client processes that generate requests and a set of servers that handle them. The server processes consume 85% of the user time, with the remaining going to the clients. Although the I/O latency is hidden by careful tuning and enough requests to keep the processor busy, the server processes typically block for I/O after about 25,000 instructions. Overall, 71% of the execution time is spent in user mode, 18% in the operating system, and 11% idle, primarily waiting for I/O. Of the commercial applications studied, the OLTP application stresses the memory system the hardest and shows significant challenges even when evaluated with much larger L3 caches. For example, on the AlphaServer, the processors are stalled

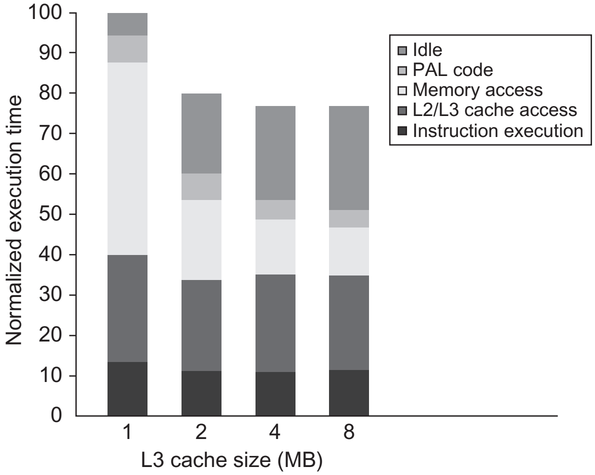


Figure 5.10 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 to 8 MiB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, more server processes are needed to cover the I/O latency. The workload could be retuned to increase the computation/communication balance, holding the idle time in check. The PAL code is a set of sequences of specialized OS-level instructions executed in privileged mode; an example is the TLB miss handler.

for approximately 90% of the cycles with memory accesses occupying almost half the stall time and L2 misses 25% of the stall time.

We start by examining the effect of varying the size of the L3 cache. In these studies, the L3 cache is varied from 1 to 8 MiB per processor; at 2 MiB per processor, the total size of L3 is equal to that of the Intel i7 6700. In the case of the i7, however, the cache is shared, which provides both some advantages and disadvantages. It is unlikely that the shared 8 MiB cache will outperform separate L3 caches with a total size of 16 MiB. Figure 5.10 shows the effect of increasing the cache size, using two-way set associative caches, which reduces the large number of conflict misses. The execution time is improved as the L3 cache grows because of the reduction in L3 misses. Surprisingly, almost all of the gain occurs in going from 1 to 2 MiB (or 4 to 8 MiB of total cache for the four processors). There is little additional gain beyond that, despite the fact that cache misses are still a cause of significant performance loss with 2 MiB and 4 MiB caches. The question is, Why?

To better understand the answer to this question, we need to determine what factors contribute to the L3 miss rate and how they change as the L3 cache grows. Figure 5.11 shows these data, displaying the number of memory access cycles contributed per instruction from five sources. The two largest sources of L3 memory access cycles with a 1 MiB L3 are instruction and capacity/conflict misses. With a larger L3, these two sources shrink to be minor contributors. Unfortunately, the

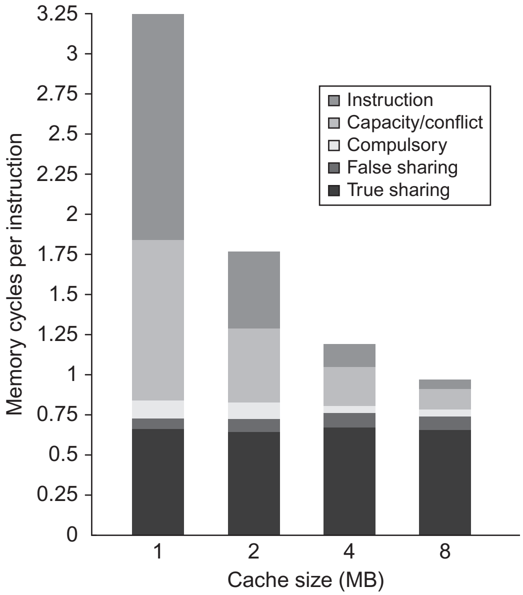


Figure 5.11 The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

compulsory, false sharing, and true sharing misses are unaffected by a larger L3. Thus, at 4 and 8 MiB, the true sharing misses generate the dominant fraction of the misses; the lack of change in true sharing misses leads to the limited reductions in the overall miss rate when increasing the L3 cache size beyond 2 MiB.

Increasing the cache size eliminates most of the uniprocessor misses while leaving the multiprocessor misses untouched. How does increasing the processor count affect different types of misses? Figure 5.12 shows these data assuming a base configuration with a 2 MiB, two-way set associative L3 cache (the same effective per processor cache size as the i7 but with less associativity). As we might expect, the increase in the true sharing miss rate, which is not compensated for by any decrease in the uniprocessor misses, leads to an overall increase in the memory access cycles per instruction.

The final question we examine is whether increasing the block size—which should decrease the instruction and cold miss rate and, within limits, also reduce the capacity/conflict miss rate and possibly the true sharing miss rate—is helpful for this workload. Figure 5.13 shows the number of misses per 1000 instructions as the block size is increased from 32 to 256 bytes. Increasing the block size from 32 to 256 bytes affects four of the miss rate components:

- The true sharing miss rate decreases by more than a factor of 2, indicating some locality in the true sharing patterns.

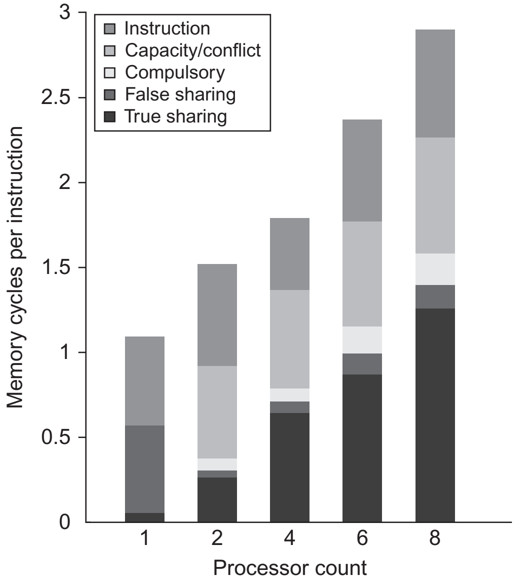


Figure 5.12 The contribution to memory access cycles increases as processor count increases primarily because of increased true sharing. The compulsory misses slightly increase because each processor must now handle more compulsory misses.

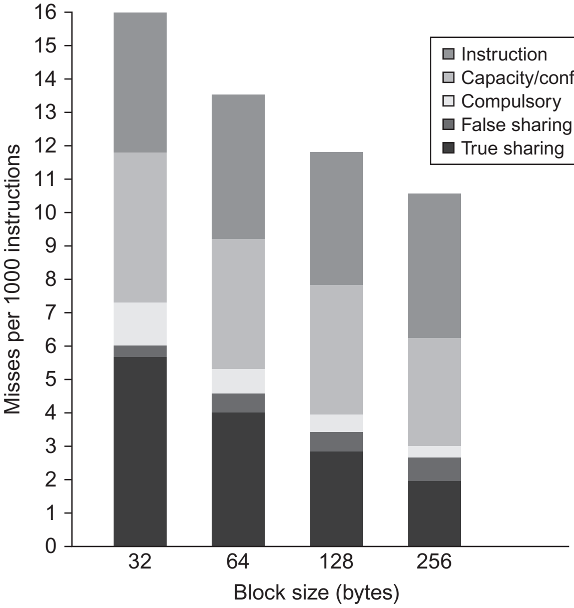


Figure 5.13 The number of misses per 1000 instructions drops steadily as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. The L3 cache is 2 MiB, two-way set associative.

- The compulsory miss rate significantly decreases, as we would expect.
- The conflict/capacity misses show a small decrease (a factor of 1.26 compared to a factor of 8 increase in block size), indicating that the spatial locality is not high in the uniprocessor misses that occur with L3 caches larger than 2 MiB.
- The false sharing miss rate, although small in absolute terms, nearly doubles.

The lack of a significant effect on the instruction miss rate is startling. If there were an instruction-only cache with this behavior, we would conclude that the spatial locality is very poor. In the case of mixed L2 and L3 caches, other effects such as instruction-data conflicts may also contribute to the high instruction cache miss rate for larger blocks. Other studies have documented the low spatial locality in the instruction stream of large database and OLTP workloads, which have lots of short basic blocks and special-purpose code sequences. Based on these data, the miss penalty for a larger block size L3 to perform as well as the 32-byte block size L3 can be expressed as a multiplier on the 32-byte block size penalty.

Block size	Miss penalty relative to 32-byte block miss penalty
64 bytes	1.19
128 bytes	1.36
256 bytes	1.52

With modern DDR SDRAMs that make block access fast, these numbers are attainable, especially at the 64 byte (the i7 block size) and the 128 byte block size. Of course, we must also worry about the effects of the increased traffic to memory and possible contention for the memory with other cores. This latter effect may easily negate the gains obtained from improving the performance of a single processor.

A Multiprogramming and OS Workload

Our next study is a multiprogrammed workload consisting of both user activity and OS activity. The workload used is two independent copies of the compile phases of the Andrew benchmark, a benchmark that emulates a software development environment. The compile phase consists of a parallel version of the UNIX “make” command executed using eight processors. The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems. The workload is run with 128 MiB of memory, and no paging activity takes place.

The workload has three distinct phases: compiling the benchmarks, which involves substantial compute activity; installing the object files in a library; and removing the object files. The last phase is completely dominated by I/O, and only two processes are active (one for each of the runs). In the middle phase, I/O also plays a major role, and the processor is largely idle. The overall workload is much more system- and I/O-intensive than the OLTP workload.

For the workload measurements, we assume the following memory and I/O systems:

- *Level 1 instruction cache*—32 KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.
- *Level 1 data cache*—32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. Our focus is on examining the behavior in the Level 1 data cache, in contrast to the OLTP study, which focused on the L3 cache.
- *Level 2 cache*—1 MiB unified, two-way set associative with a 128-byte block, 10 clock cycle hit time.
- *Main memory*—Single memory on a bus with an access time of 100 clock cycles.
- *Disk system*—Fixed-access latency of 3 ms (less than normal to reduce idle time).

Figure 5.14 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken down into four components:

1. *Idle*—Execution in the kernel mode idle loop
2. *User*—Execution in user code
3. *Synchronization*—Execution or waiting for synchronization variables
4. *Kernel*—Execution in the OS that is neither idle nor in synchronization access

This multiprogramming workload has a significant instruction cache performance loss, at least for the OS. The instruction cache miss rate in the OS for a 64-byte block size, two-way set associative cache varies from 1.7% for a 32 KB

	User execution	Kernel execution	Synchronization wait	Processor idle (waiting for I/O)
Instructions executed	27%	3%	1%	69%
Execution time	27%	7%	2%	64%

Figure 5.14 The distribution of execution time in the multiprogrammed parallel "make" workload. The high fraction of idle time is due to disk latency when only one of the eight processors is active. These data and the subsequent measurements for this workload were collected with the SimOS system (Rosenblum et al., 1995). The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University.

cache to 0.2% for a 256 KB cache. User-level instruction cache misses are roughly one-sixth of the OS rate, across the variety of cache sizes. This partially accounts for the fact that, although the user code executes nine times as many instructions as the kernel, those instructions take only about four times as long as the smaller number of instructions executed by the kernel.

Performance of the Multiprogramming and OS Workload

In this section, we examine the cache performance of the multiprogrammed workload as the cache size and block size are changed. Because of differences between the behavior of the kernel and that of the user processes, we keep these two components separate. Remember, though, that the user processes execute more than eight times as many instructions, so the overall miss rate is determined primarily by the miss rate in user code, which as we will see, is often one-fifth of the kernel miss rate.

Although the user code executes more instructions, the behavior of the operating system can cause more cache misses than the user processes for two reasons beyond larger code size and lack of locality. First, the kernel initializes all pages before allocating them to a user, which significantly increases the compulsory component of the kernel's miss rate. Second, the kernel actually shares data and thus has a nontrivial coherence miss rate. In contrast, user processes cause coherence misses only when the process is scheduled on a different processor, and this component of the miss rate is small. This is a major difference between a multiprogrammed workload and one like the OLTP workload.

Figure 5.15 shows the data miss rate versus data cache size and versus block size for the kernel and user components. Increasing the data cache size affects the user miss rate more than it affects the kernel miss rate. Increasing the block size has beneficial effects for both miss rates because a larger fraction of the misses arise from compulsory and capacity, both of which can be potentially improved with larger block sizes. Because coherence misses are relatively rarer, the negative effects of increasing block size are small. To understand why the kernel and user processes behave differently, we can look at how the kernel misses behave.

Figure 5.16 shows the variation in the kernel misses versus increases in cache size and in block size. The misses are broken into three classes: compulsory misses, coherence misses (from both true and false sharing), and capacity/conflict misses (which include misses caused by interference between the OS and the user process and between multiple user processes). Figure 5.16 confirms that, for the kernel references, increasing the cache size reduces only the uniprocessor capacity/conflict miss rate. In contrast, increasing the block size causes a reduction in the compulsory miss rate. The absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are probably insignificant, although such misses may be offsetting some of the gains from reducing the true sharing misses.

If we examine the number of bytes needed per data reference, as in Figure 5.17, we see that the kernel has a higher traffic ratio that grows with block size. It is easy to see why this occurs: when going from a 16-byte block to a 128-byte block, the

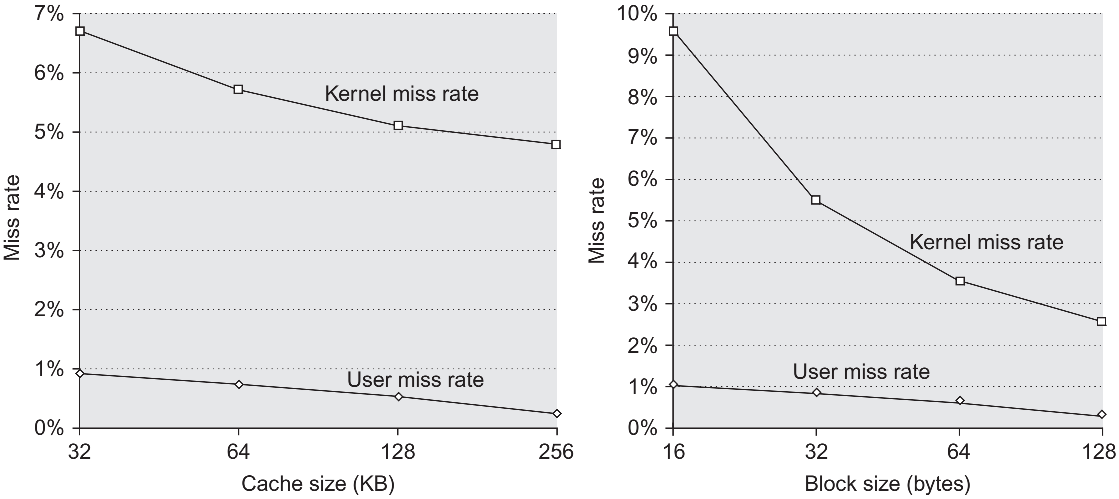


Figure 5.15 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 to 256 KB (with a 32-byte block) causes the user miss rate to decrease proportionately more than the kernel miss rate: the user-level miss rate drops by almost a factor of 3, whereas the kernel-level miss rate drops by a factor of only 1.3. At the largest size, the L1 is closer to the size of L2 in a modern multicore processors. Thus the data indicates that the kernel miss rate will still be significant in an L2 cache. The miss rate for both user and kernel components drops steadily as the L1 block size is increased (while keeping the L1 cache at 32 KB). In contrast to the effects of increasing the cache size, increasing the block size improves the kernel miss rate more significantly (just under a factor of 4 for the kernel references when going from 16-byte to 128-byte blocks versus just under a factor of 3 for the user references).

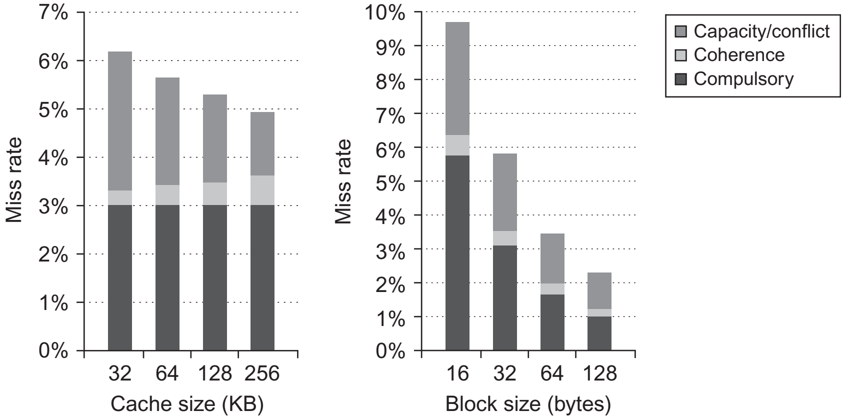


Figure 5.16 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant because it is unaffected by cache size. The capacity component drops by more than a factor of 2, whereas the coherence component nearly doubles. The increase in coherence misses occurs because the probability of a miss being caused by an invalidation increases with cache size, since fewer entries are bumped due to capacity. As we would expect, the increasing block size of the L1 data cache substantially reduces the compulsory miss rate in the kernel references. It also has a significant impact on the capacity miss rate, decreasing it by a factor of 2.4 over the range of block sizes. The increased block size has a small reduction in coherence traffic, which appears to stabilize at 64 bytes, with no change in the coherence miss rate in going to 128-byte lines. Because there are no significant reductions in the coherence miss rate as the block size increases, the fraction of the miss rate caused by coherence grows from about 7% to about 15%.

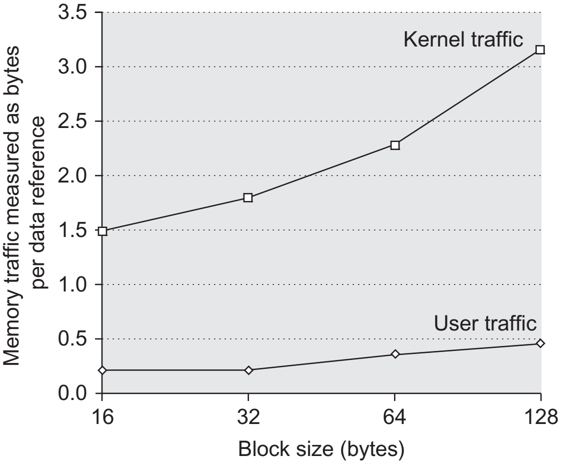


Figure 5.17 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart with the data on scientific programs shown in Appendix I.

miss rate drops by about 3.7, but the number of bytes transferred per miss increases by 8, so the total miss traffic increases by just over a factor of 2. The user program also more than doubles as the block size goes from 16 to 128 bytes, but it starts out at a much lower level.

For the multiprogrammed workload, the OS is a much more demanding user of the memory system. If more OS or OS-like activity is included in the workload, and the behavior is similar to what was measured for this workload, it will become very difficult to build a sufficiently capable memory system. One possible route to improving performance is to make the OS more cache-aware through either better programming environments or through programmer assistance. For example, the OS reuses memory for requests that arise from different system calls. Despite the fact that the reused memory will be completely overwritten, the hardware, not recognizing this, will attempt to preserve coherency and the possibility that some portion of a cache block may be read, even if it is not. This behavior is analogous to the reuse of stack locations on procedure invocations. The IBM Power series has support to allow the compiler to indicate this type of behavior on procedure invocations, and the newest AMD processors have similar support. It is harder to detect such behavior by the OS, and doing so may require programmer assistance, but the payoff is potentially even greater.

OS and commercial workloads pose tough challenges for multiprocessor memory systems, and unlike scientific applications, which we examine in Appendix I, they are less amenable to algorithmic or compiler restructuring. As the number of cores increases, predicting the behavior of such applications is likely to get more difficult. Emulation or simulation methodologies that allow the simulation of tens to hundreds of cores with large applications (including operating systems) will be crucial to maintaining an analytical and quantitative approach to design.

5.4 Distributed Shared-Memory and Directory-Based Coherence

As we saw in Section 5.2, a snooping protocol requires communication with all caches on every cache miss, including writes of potentially shared data. The absence of any centralized data structure that tracks the state of the caches is both the fundamental advantage of a snooping-based scheme, since it allows it to be inexpensive, as well as its Achilles' heel when it comes to scalability.

For example, consider a multiprocessor consisting of four 4-core multicores capable of sustaining one data reference per clock and a 4 GHz clock. From the data in Section I.5 in Appendix I, we can see that the applications may require 4–170 GiB/s of memory bus bandwidth. The maximum memory bandwidth supported by the i7 with two DDR4 memory channels is 34 GiB/s. If several i7 multicore processors shared the same memory system, they would easily swamp it. In the last few years, the development of multicore processors forced all designers to shift to some form of distributed memory to support the bandwidth demands of the individual processors.

We can increase the memory bandwidth and interconnection bandwidth by distributing the memory, as shown in Figure 5.2 on page 373; this immediately separates local memory traffic from remote memory traffic, reducing the bandwidth demands on the memory system and on the interconnection network. Unless we eliminate the need for the coherence protocol to broadcast on every cache miss, distributing the memory will gain us little.

As we mentioned earlier, the alternative to a snooping-based coherence protocol is a *directory protocol*. A directory keeps the state of every block that may be cached. Information in the directory includes which caches (or collections of caches) have copies of the block, whether it is dirty, and so on. Within a multicore with a shared outermost cache (say, L3), it is easy to implement a directory scheme: simply keep a bit vector of the size equal to the number of cores for each L3 block. The bit vector indicates which private L2 caches may have copies of a block in L3, and invalidations are sent only to those caches. This works perfectly for a single multicore if L3 is inclusive, and this scheme is the one used in the Intel i7.

The solution of a single directory used in a multicore is not scalable, even though it avoids broadcast. The directory must be distributed, but the distribution must be done in a way that the coherence protocol knows where to find the directory information for any cached block of memory. The obvious solution is to distribute the directory along with the memory so that different coherence requests can go to different directories, just as different memory requests go to different memories. If the information is maintained at an outer cache, like L3, which is multibanked, the directory information can be distributed with the different cache banks, effectively increasing the bandwidth.

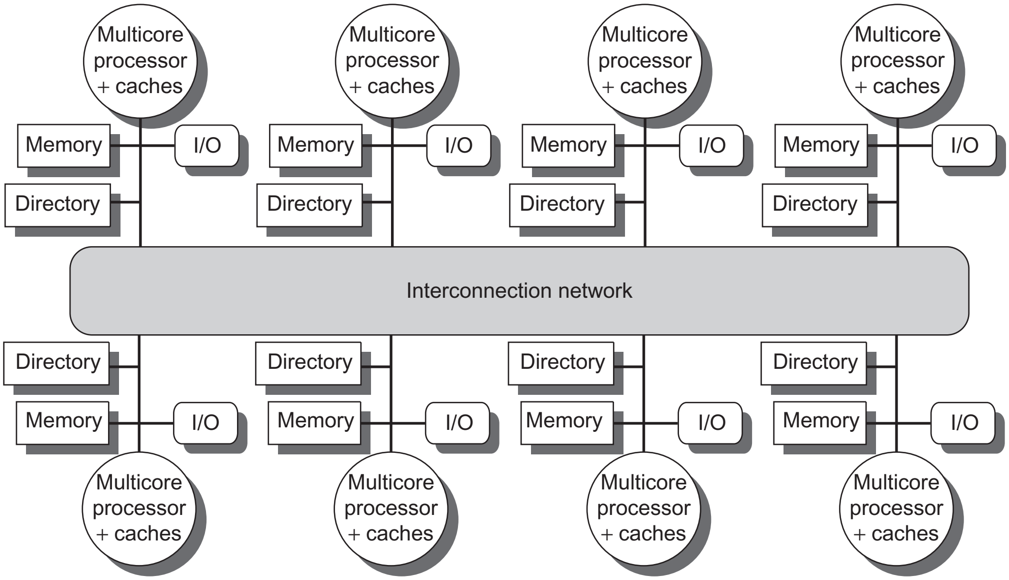


Figure 5.18 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case, a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will handle both the maintenance of the directory information and any coherence actions needed within the multicore node.

A distributed directory retains the characteristic that the sharing status of a block is always in a single known location. This property, together with the maintenance of information that says what other nodes may be caching the block, is what allows the coherence protocol to avoid broadcast. Figure 5.18 shows how our distributed-memory multiprocessor looks with the directories added to each node.

The simplest directory implementations associate an entry in the directory with each memory block. In such implementations, the amount of information is proportional to the product of the number of memory blocks (where each block is the same size as the L2 or L3 cache block) times the number of nodes, where a node is a single multicore processor or a small collection of processors that implements coherence internally. This overhead is not a problem for multiprocessors with less than a few hundred processors (each of which might be a multicore) because the directory overhead with a reasonable block size will be tolerable. For larger multiprocessors, we need methods to allow the directory structure to be efficiently scaled, but only supercomputer-sized systems need to worry about this.

Directory-Based Cache Coherence Protocols: The Basics

Just as with a snooping protocol, there are two primary operations that a directory protocol must implement: handling a read miss and handling a write to a shared, clean cache block. (Handling a write miss to a block that is currently shared is a simple combination of these two.) To implement these operations, a directory must track the state of each cache block. In a simple protocol, these states could be the following:

- *Shared*—One or more nodes have the block cached, and the value in memory is up to date (as well as in all the caches).
- *Uncached*—No node has a copy of the cache block.
- *Modified*—Exactly one node has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the *owner* of the block.

In addition to tracking the state of each potentially shared memory block, we must track which nodes have copies of that block because those copies will need to be invalidated on a write. The simplest way to do this is to keep a bit vector for each memory block. When the block is shared, each bit of the vector indicates whether the corresponding processor chip (which is likely a multicore) has a copy of that block. We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state. For efficiency reasons, we also track the state of each cache block at the individual caches.

The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different. The processes of invalidating and locating an exclusive copy of a data item are different because they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes. In a snooping protocol, these two steps are combined through the use of a broadcast to all the nodes.

Before we see the protocol state diagrams, it is useful to examine a catalog of the message types that may be sent between the processors and the directories for the purpose of handling misses and maintaining coherence. Figure 5.19 shows the types of messages sent among nodes. The *local node* is the node where a request originates. The *home node* is the node where the memory location and the directory entry of an address reside. The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known. For example, the high-order bits may provide the node number, whereas the low-order bits provide the offset within the memory on that node. The local node may also be the home node. The directory must be accessed when the home node is the local node because copies may exist in yet a third node, called a *remote node*.

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

Figure 5.19 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The fourth through sixth messages are messages sent to a remote node by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write-backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol because any dirty block must be exclusive and any shared block is always available in the home memory.

A remote node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared. A remote node may be the same as either the local node or the home node. In such cases, the basic protocol does not change, but interprocessor messages may be replaced with intraprocessor messages.

In this section, we assume a simple model of memory consistency. To minimize the type of messages and the complexity of the protocol, we make an assumption that messages will be received and acted upon in the same order they are sent. This assumption may not be true in practice and can result in additional complications, some of which we address in Section 5.6 when we discuss memory consistency models. In this section, we use this assumption to ensure that invalidates sent by a node are honored before new messages are transmitted, just as we assumed in the discussion of implementing snooping protocols. As we did in the snooping case, we omit some details necessary to implement the coherence protocol. In particular, the serialization of writes and knowing that the invalidates for a write have completed are not as simple as in the broadcast-based snooping mechanism. Instead, explicit acknowledgments are required in response to write misses and invalidate requests. We discuss these issues in more detail in Appendix I.

An Example Directory Protocol

The basic states of a cache block in a directory-based protocol are exactly like those in a snooping protocol, and the states in the directory are also analogous to those we showed earlier. Thus we can start with simple state diagrams that show the state transitions for an individual cache block and then examine the state diagram for the directory entry corresponding to each block in memory. As in the snooping case, these state transition diagrams do not represent all the details of a coherence protocol; however, the actual controller is highly dependent on a number of details of the multiprocessor (message delivery properties, buffering structures, and so on). In this section, we present the basic protocol state diagrams. The knotty issues involved in implementing these state transition diagrams are examined in Appendix I.

Figure 5.20 shows the protocol actions to which an individual cache responds. We use the same notation as in the last section, with requests coming from outside the node in gray and actions in bold. The state transitions for an individual cache are caused by read misses, write misses, invalidates, and data fetch requests; Figure 5.20 shows these operations. An individual cache also generates read miss, write miss, and invalidate messages that are sent to the home directory. Read and write misses require data value replies, and these events wait for replies before changing state. Knowing when invalidates complete is a separate problem and is handled separately.

The operation of the state transition diagram for a cache block in Figure 5.20 is essentially the same as it is for the snooping case: the states are identical, and the stimulus is almost identical. The write miss operation, which was broadcast on the bus (or other network) in the snooping scheme, is replaced by the data fetch and invalidate operations that are selectively sent by the directory controller. Like the snooping protocol, any cache block must be in the exclusive state when it is written, and any shared block must be up to date in memory. In many multicore processors, the outermost level in the processor cache is shared among the cores (as is the L3 in the Intel i7, the AMD Opteron, and the IBM Power7), and hardware at that level maintains coherence among the private caches of each core on the same chip, using either an internal directory or snooping. Thus the on-chip multicore coherence mechanism can be used to extend coherence among a larger set of processors simply by interfacing to the outermost shared cache. Because this interface is at L3, contention between the processor and coherence requests is less of an issue, and duplicating the tags could be avoided.

In a directory-based protocol, the directory implements the other half of the coherence protocol. A message sent to a directory causes two different types of

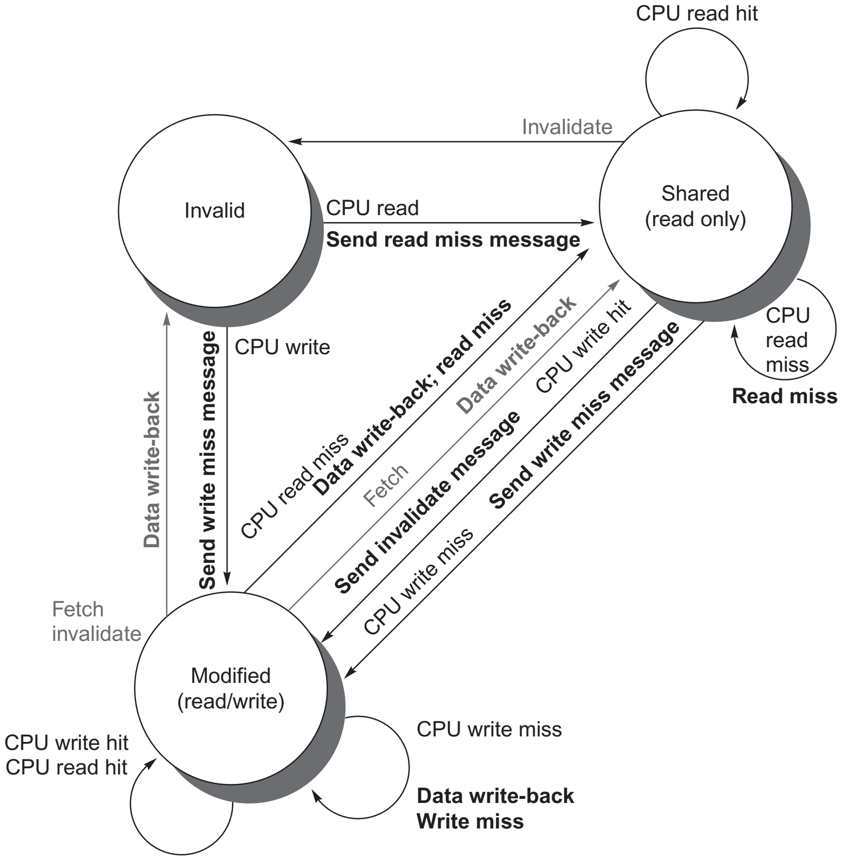

CPU read hit
Invalidate
Invalid
CPU read
Send read miss message
Shared
(read only)
CPU
read
miss
Read miss
CPU write
Send write miss message
Data write-back
Data write-back; read miss
CPU read miss
Data write-back
Fetch
CPU write hit
Send invalidate message
Send write miss message
CPU write miss
Fetch
invalidate
Modified
(read/write)
CPU write hit
CPU read hit
CPU write miss
Data write-back
Write miss

Figure 5.20 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in *black*, and those from the home directory are shown in *gray*. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

actions: updating the directory state and sending additional messages to satisfy the request. The states in the directory represent the three standard states for a block; unlike in a snooping scheme, however, the directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block.

The memory block may be uncached by any node, cached in multiple nodes and readable (shared), or cached exclusively and writable in exactly one node. In addition to the state of each block, the directory must track the set of nodes that have a copy of a block; we use a set called *Sharers* to perform this function. In multiprocessors with fewer than 64 nodes (each of which may represent four to eight times as many processors), this set is typically kept as a bit vector. Directory

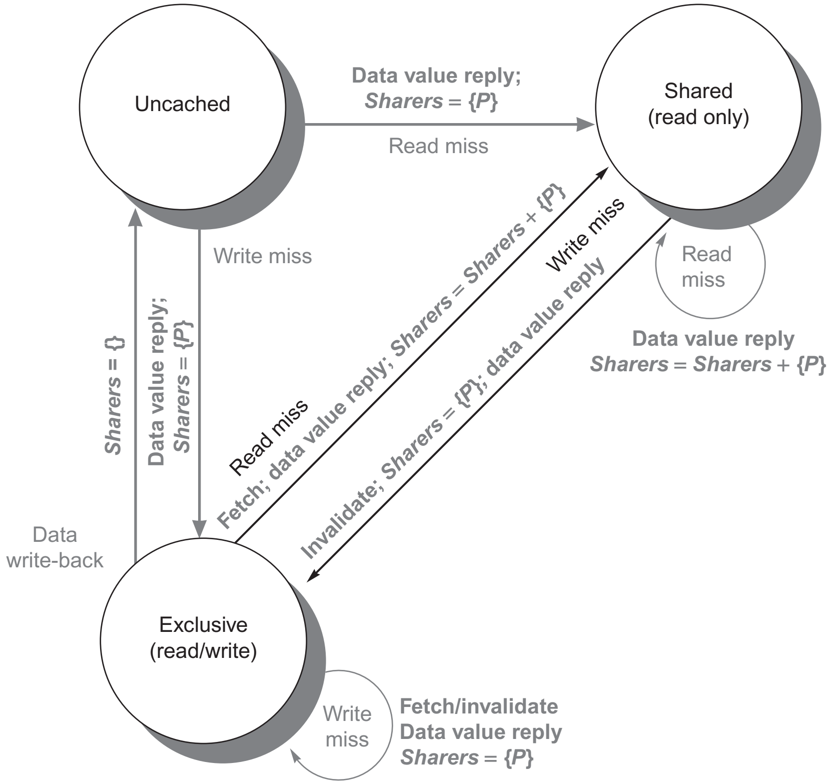

Uncached
Shared
(read only)
Exclusive
(read/write)
Data value reply;
Sharers = {P}
Read miss
Write miss
Read miss
Fetch; data value reply; Sharers = Sharers + {P}
Write miss
Invalidate; Sharers = {P}; data value reply
Read
miss
Data value reply
Sharers = Sharers + {P}
Sharers = {}
Data value reply;
Sharers = {P}
Data
write-back
Write
miss
Fetch/invalidate
Data value reply
Sharers = {P}

Figure 5.21 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. *Bold* indicates the action taken by the directory in response to the request.

requests need to update the set Sharers and also read the set to perform invalidations.

Figure 5.21 shows the actions taken at the directory in response to messages received. The directory receives three different requests: read miss, write miss, and data write-back. The messages sent in response by the directory are shown in bold, and the updating of the set Sharers is shown in bold italics. Because all the stimulus messages are external, all actions are shown in gray. Our simplified protocol assumes that some actions are atomic, such as requesting a value and sending it to another node; a realistic implementation cannot use this assumption.

To understand these directory operations, let's examine the requests received and actions taken state by state. When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are

- *Read miss*—The requesting node is sent the requested data from memory, and the requester is made the only sharing node. The state of the block is made shared.

- *Write miss*—The requesting node is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

- *Read miss*—The requesting node is sent the requested data from memory, and the requesting node is added to the sharing set.
- *Write miss*—The requesting node is sent the value. All nodes in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting node. The state of the block is made exclusive.

When the block is in the exclusive state, the current value of the block is held in a cache on the node identified by the set Sharers (the owner), so there are three possible directory requests:

- *Read miss*—The owner is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting node is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
- *Data write-back*—The owner is replacing the block and therefore must write it back. This write-back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
- *Write miss*—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting node, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

This state transition diagram in Figure 5.21 is a simplification, just as it was in the snooping cache case. In the case of a directory, as well as a snooping scheme implemented with a network other than a bus, our protocols will need to deal with nonatomic memory transactions. Appendix I explores these issues in depth.

The directory protocols used in real multiprocessors contain additional optimizations. In particular, in this protocol when a read or write miss occurs for a block that is exclusive, the block is first sent to the directory at the home node. From there it is stored into the home memory and also sent to the original requesting node. Many of the protocols in use in commercial multiprocessors forward the data from the owner node to the requesting node directly (as well as performing the write-back to the home). Such optimizations often add complexity by increasing the possibility of deadlock and by increasing the types of messages that must be handled.

Implementing a directory scheme requires solving most of the same challenges we discussed for snooping protocols. There are, however, new and additional problems, which we describe in Appendix I. In Section 5.8, we briefly describe how modern multicores extend coherence beyond a single chip. The combinations of multichip coherence and multicore coherence include all four possibilities of snooping/snooping (AMD Opteron), snooping/directory, directory/snooping, and directory/directory! Many multiprocessors have chosen some form of snooping within a single chip, which is attractive if the outermost cache is shared and inclusive, and directories across multiple chips. Such an approach simplifies implementation because only the processor chip, rather than an individual core, need be tracked.

5.5 Synchronization: The Basics

Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions. For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value. Software synchronization mechanisms are then constructed using this capability. In this section, we focus on the implementation of lock and unlock synchronization operations. Lock and unlock can be used straightforwardly to create mutual exclusion, as well as to implement more complex synchronization mechanisms.

In high-contention situations, synchronization can become a performance bottleneck because contention introduces additional delays and because latency is potentially greater in such a multiprocessor. We discuss how the basic synchronization mechanisms of this section can be extended for large processor counts in Appendix I.

Basic Hardware Primitives

The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location. Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases. There are a number of alternative formulations of the basic hardware primitives, all of which provide the ability to atomically read and modify a location, together with some way to tell whether the read and write were performed atomically. These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers. In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library, a process that is often complex and tricky. Let’s start with one such hardware primitive and show how it can be used to build some basic synchronization operations.

One typical operation for building synchronization operations is the *atomic exchange*, which interchanges a value in a register for a value in memory. To see how to use this to build a basic synchronization operation, assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock. The value returned from the exchange instruction is 1 if some other processor had otherwise already claimed access and 0. In the latter case, the value is also changed to 1, preventing any competing exchange from also retrieving a 0.

For example, consider two processors that each try to do the exchange simultaneously: This race is broken because exactly one of the processors will perform the exchange first, returning 0, and the second processor will return 1 when it does the exchange. The key to using the exchange (or swap) primitive to implement synchronization is that the operation is atomic: the exchange is indivisible, and two simultaneous exchanges will be ordered by the write serialization mechanisms. It is impossible for two processors trying to set the synchronization variable in this manner to both determine they have simultaneously set the variable.

There are a number of other atomic primitives that can be used to implement synchronization. They all have the key property that they read and update a memory value in such a manner that we can tell whether the two operations executed atomically. One operation, present in many older multiprocessors, is *test-and-set*, which tests a value and sets it if the value passes the test. For example, we could define an operation that tested for 0 and set the value to 1, which can be used in a fashion similar to how we used atomic exchange. Another atomic synchronization primitive is *fetch-and-increment*: it returns the value of a memory location and atomically increments it. By using the value 0 to indicate that the synchronization variable is unclaimed, we can use fetch-and-increment, just as we used exchange. There are other uses of operations like fetch-and-increment, which we will see shortly.

Implementing a single atomic memory operation introduces some challenges because it requires both a memory read and a write in a single, uninterruptible instruction. This requirement complicates the implementation of coherence because the hardware cannot allow any other operations between the read and the write, and yet must not deadlock.

An alternative is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as though the instructions were atomic. The pair of instructions is effectively atomic if it appears as though all other operations executed by any processor occurred before or after the pair. Thus, when an instruction pair is effectively atomic, no other processor can change the value between the instruction pair. This is the approach used in the MIPS processors and in RISC V.

In RISC V, the pair of instructions includes a special load called a *load reserved* (also called load linked or *load locked)* and a special store called a *store conditional*. Load reserved loads the contents of memory given by rs1 into rd and creates a reservation on that memory address. Store conditional stores the value in rs2 into the memory address given by rs1. If the reservation of the load is broken by a write to the same memory location, the store conditional fails and writes a nonzero to rd; if it succeeds, the store conditional writes 0. If the processor does a context switch between the two instructions, then the store conditional always fails.

These instructions are used in sequence, and because the load reserved returns the initial value and the store conditional returns 0 only if it succeeds, the following sequence implements an atomic exchange on the memory location specified by the contents of x1 with the value in x4:

```
try:    mov     x3,x4       ;mov exchange value
        lr      x2,x1       ;load reserved from
        sc      x3,0(x1)    ;store conditional
        bnez    x3,try      ;branch store fails
        mov     x4,x2       ;put load value in x4
```

At the end of this sequence, the contents of `x4` and the memory location specified by `x1` have been atomically exchanged. Anytime a processor intervenes and modifies the value in memory between the `lr` and `sc` instructions, the `sc` returns 0 in `x3`, causing the code sequence to try again.

An advantage of the load reserved/store conditional mechanism is that it can be used to build other synchronization primitives. For example, here is an atomic fetch-and-increment:

```
try:    lr     x2,x1       ;load reserved 0(x1)
        addi   x3,x2,1     ;increment
        sc     x3,0(x1)    ;store conditional
        bnez   x3,try      ;branch store fails
```

These instructions are typically implemented by keeping track of the address specified in the `lr` instruction in a register, often called the *reserved register*. If an interrupt occurs, or if the cache block matching the address in the link register is invalidated (e.g., by another `sc`), the link register is cleared. The `sc` instruction simply checks that its address matches that in the reserved register. If so, the `sc` succeeds; otherwise, it fails. Because the store conditional will fail after either another attempted store to the load reserved address or any exception, care must be taken in choosing what instructions are inserted between the two instructions. In particular, only register-register instructions can safely be permitted; otherwise, it is possible to create deadlock situations where the processor can never complete the `sc`. In addition, the number of instructions between the load reserved and the store conditional should be small to minimize the probability that either an unrelated event or a competing processor causes the store conditional to fail frequently.

Implementing Locks Using Coherence

Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks*—locks that a processor continuously tries to acquire, spinning around a loop until it succeeds. Spin locks are used when programmers expect the lock to be held for a very short amount of time and when they want the process of locking to be low latency when the lock is available. Because spin locks tie up the processor waiting in a loop for the lock to become free, they are inappropriate in some circumstances.

The simplest implementation, which we would use if there were no cache coherence, would be to keep the lock variables in memory. A processor could continually try to acquire the lock using an atomic operation, say, atomic exchange, and test whether the exchange returned the lock as free. To release the lock, the processor simply stores the value 0 to the lock. Here is the code sequence to lock a spin lock whose address is in `x1`. It uses `EXCH` as a macro for the atomic exchange sequence from page 414:

```
         addi  x2,R0,#1
lockit:  EXCH  x2,0(x1)      ;atomic exchange
         bnez  x2,lockit     ;already locked?
```

If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently. Caching locks has two advantages. First, it allows an implementation where the process of "spinning" (trying to test and acquire the lock in a tight loop) could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock. The second advantage comes from the observation that there is often locality in lock accesses; that is, the processor that used the lock last will use it again in the near future. In such cases, the lock value may reside in the cache of that processor, greatly reducing the time to acquire the lock.

Obtaining the first advantage—being able to spin on a local cached copy rather than generating a memory request for each attempt to acquire the lock—requires a change in our simple spin procedure. Each attempt to exchange in the preceding loop requires a write operation. If multiple processors are attempting to get the lock, each will generate the write. Most of these writes will lead to write misses because each processor is trying to obtain the lock variable in an exclusive state.

Thus we should modify our spin lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available. Then it attempts to acquire the lock by doing a swap operation. A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is unlocked. The processor then races against all other processes that were similarly "spin waiting" to see which can lock the variable first. All processes use a swap instruction that reads the old value and stores a 1 into the lock variable. The single winner will see the 0, and the losers will see a 1 that was placed there by the winner. (The losers will continue to set the variable to the locked value, but that doesn't matter.) The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock, which starts the race all over again. Here is the code to perform this spin lock (remember that 0 is unlocked and 1 is locked):

```
lockit:  ld    x2,0(x1)    ;load of lock
         bnez  x2,lockit   ;not available-spin
         addi  x2,R0,#1    ;load locked value
         EXCH  x2,0(x1)    ;swap
         bnez  x2,lockit   ;branch if lock wasn't 0
```

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied.
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied.
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Figure 5.22 Cache coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write invalidate coherence. P0 starts with the lock (step 1), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3–5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails, so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than 8 clock ticks because acquiring the bus and replying to misses take much longer. Once step 8 is reached, the process can repeat with P2, eventually getting exclusive access and setting the lock to 0.

Let's examine how this "spin lock" scheme uses the cache coherence mechanisms. Figure 5.22 shows the processor and bus or directory operations for multiple processes trying to lock a variable using an atomic swap. Once the processor with the lock stores a 0 into the lock, all other caches are invalidated and must fetch the new value to update their copy of the lock. One such cache gets the copy of the unlocked value (0) first and performs the swap. When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning.

This example shows another advantage of the load reserved/store conditional primitives: the read and write operations are explicitly separated. The load reserved need not cause any bus traffic. This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (`x1` has the address of the lock, the `lr` has replaced the `LD`, and the `sc` has replaced the `EXCH`):

```
lockit: lr    x2,0(x1)   ;load reserved
        bnez  x2,lockit  ;not available-spin
        addi  x2,R0,#1   ;locked value
        sc    x2,0(x1)   ;store
        bnez  x2,lockit  ;branch if store fails
```

The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

5.6 Models of Memory Consistency: An Introduction

Cache coherence ensures that multiple processors see a consistent view of memory. It does not answer the question of *how* consistent the view of memory must be. By "how consistent," we are really asking when a processor must see a value that has been updated by another processor. Because processors communicate through shared variables (used both for data values and for synchronization), the question boils down to this: In what order must a processor observe the data writes of another processor? Because the only way to "observe the writes of another processor" is through reads, the question becomes what properties must be enforced among reads and writes to different locations by different processors?

Although the question of how consistent memory must be seems simple, it is remarkably complicated, as we can see with a simple example. Here are two code segments from processes `P1` and `P2`, shown side by side:

```
P1:     A = 0;            P2:   B = 0;
        .....                   .....
        A = 1;                  B = 1;
L1:     if (B == 0)...    L2:   if (A == 0)...
```

Assume that the processes are running on different processors, and that locations `A` and `B` are originally cached by both processors with the initial value of 0. If writes always take immediate effect and are immediately seen by other processors, it will be impossible for *both* IF statements (labeled `L1` and `L2`) to evaluate their conditions as true, since reaching the IF statement means that either `A` or `B` must have been assigned the value 1. But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay. Then it is possible that both `P1` and `P2` have not seen the invalidations for `B` and `A` (respectively) *before* they attempt to read the values. The question now is should this behavior be allowed, and, if so, under what conditions?

The most straightforward model for memory consistency is called *sequential consistency*. Sequential consistency requires that the result of any execution be the same as though the memory accesses executed by each processor were kept in order and the accesses among different processors were arbitrarily interleaved. Sequential consistency eliminates the possibility of some nonobvious execution in the previous example because the assignments must be completed before the IF statements are initiated.

The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed. Of course, it is equally effective to delay the next memory access until the previous one is completed. Remember that memory consistency involves operations among different variables: the two accesses that must be ordered are actually to different memory locations. In our example, we must delay the read of `A` or `B` (`A == 0` or `B == 0`) until the previous write has completed (`B = 1` or `A = 1`). Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read.

Although sequential consistency presents a simple programming paradigm, it reduces potential performance, especially in a multiprocessor with a large number of processors or long interconnect delays, as we can see in the following example.

Example Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the coherence controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

Answer When we wait for invalidates, each write takes the sum of the ownership time plus the time to complete the invalidates. Because the invalidates can overlap, we need only worry about the last one, which starts $10+10+10+10=40$ cycles after ownership is established. Therefore the total time for the write is $50+40+80=170$ cycles. In comparison, the ownership time is only 50 cycles. With appropriate write buffer implementations, it is even possible to continue before ownership is established.

To provide better performance, researchers and architects have explored two different routes. First, they developed ambitious implementations that preserve sequential consistency but use latency-hiding techniques to reduce the penalty; we discuss these in Section 5.7. Second, they developed less restrictive memory consistency models that allow for faster hardware. Such models can affect how the programmer sees the multiprocessor, so before we discuss these less restrictive models, let's look at what the programmer expects.

The Programmer's View

Although the sequential consistency model has a performance disadvantage, from the viewpoint of the programmer, it has the advantage of simplicity. The challenge is to develop a programming model that is simple to explain and yet allows a high-performance implementation.

One such programming model that allows us to have a more efficient implementation is to assume that programs are *synchronized*. A program is synchronized if all accesses to shared data are ordered by synchronization operations. A data reference is ordered by a synchronization operation if, in every possible execution, a write of a variable by one processor and an access (either a read or a write) of that variable by another processor are separated by a pair of synchronization operations, one executed after the write by the writing processor and one executed before the access by the second processor. Cases where variables may be updated without ordering by synchronization are called *data races* because the execution outcome depends on the relative speed of the processors, and like races in hardware design, the outcome is unpredictable, which leads to another name for synchronized programs: *data-race-free*.

As a simple example, consider a variable being read and updated by two different processors. Each processor surrounds the read and update with a lock and an unlock, both to ensure mutual exclusion for the update and to ensure that the read is consistent. Clearly, every write is now separated from a read by the other processor by a pair of synchronization operations: one unlock (after the write) and one lock (before the read). Of course, if two processors are writing a variable with no intervening reads, then the writes must also be separated by synchronization operations.

It is a broadly accepted observation that most programs are synchronized. This observation is true primarily because, if the accesses were unsynchronized, the behavior of the program would likely be unpredictable because the speed of execution would determine which processor won a data race and thus affect the results of the program. Even with sequential consistency, reasoning about such programs is very difficult.

Programmers could attempt to guarantee ordering by constructing their own synchronization mechanisms, but this is extremely tricky, can lead to buggy programs, and may not be supported architecturally, meaning that they may not work in future generations of the multiprocessor. Instead, almost all programmers will choose to use synchronization libraries that are correct and optimized for the multiprocessor and the type of synchronization.

Finally, the use of standard synchronization primitives ensures that even if the architecture implements a more relaxed consistency model than sequential consistency, a synchronized program will behave as though the hardware implemented sequential consistency.

Relaxed Consistency Models: The Basics and Release Consistency

The key idea in relaxed consistency models is to allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering so that a synchronized program behaves as though the processor were sequentially consistent. There are a variety of relaxed models that are classified according to what read and write orderings they relax. We specify the orderings by a set of rules of the form X → Y, meaning that operation X must complete before operation Y is done. Sequential consistency requires maintaining all four possible orderings: R → W, R → R, W → R, and W → W. The relaxed models are defined by the subset of four orderings they relax:

1. Relaxing only the W → R ordering yields a model known as *total store ordering* or *processor consistency*. Because this model retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization.
2. Relaxing both the W → R ordering and the W → W ordering yields a model known as *partial store order*.
3. Relaxing all four orderings yields a variety of models including *weak ordering*, the PowerPC consistency model, and *release consistency*, the RISC V consistency model.

By relaxing these orderings, the processor may obtain significant performance advantages, which is the reason that RISC V, ARMv8, as well as the C++ and C language standards chose release consistency as the model.

Release consistency distinguishes between synchronization operations that are used to *acquire* access to a shared variable (denoted S_A) and those that *release* an object to allow another processor to acquire access (denoted S_R). Release consistency is based on the observation that in synchronized programs, an acquire operation must precede a use of shared data, and a release operation must follow any updates to shared data and also precede the time of the next acquire. This property allows us to slightly relax the ordering by observing that a read or write that precedes an acquire need not complete before the acquire, and also that a read or write that follows a release need not wait for the release. Thus the orderings that are preserved involve only S_A and S_R, as shown in Figure 5.23; as the example in Figure 5.24 shows, this model imposes the fewest orders of the five models.

Release consistency provides one of the least restrictive models that is easily checkable and ensures that synchronized programs will see a sequentially consistent execution. Although most synchronization operations are either an acquire or a release (an acquire normally reads a synchronization variable and atomically updates it, and a release usually just writes it), some operations, such as a barrier, act as both an acquire and a release and cause the ordering to be equivalent to weak ordering. Although synchronization operations always ensure that previous writes have completed, we may want to guarantee that writes are completed without an identified synchronization operation. In such cases, an explicit instruction, called FENCE in RISC V, is used to ensure that all previous instructions in that thread have completed, including completion of all memory writes and associated invalidates. For more information about the complexities, implementation issues, and

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Partial store order	SPARC	$R \rightarrow R$, $R \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Weak ordering	PowerPC		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

Figure 5.23 The orderings imposed by various consistency models are shown for both ordinary accesses and synchronization accesses. The models grow from most restrictive (sequential consistency) to least restrictive (release consistency), allowing increased flexibility in the implementation. The weaker models rely on fences created by synchronization operations, as opposed to an implicit fence at every memory operation. S_A and S_R stand for acquire and release operations, respectively, and are needed to define release consistency. If we were to use the notation S_A and S_R for each S consistently, each ordering with one S would become two orderings (e.g., $S \rightarrow W$ becomes $S_A \rightarrow W$, $S_R \rightarrow W$), and each $S \rightarrow S$ would become the four orderings shown in the last line of the bottom-right table entry.

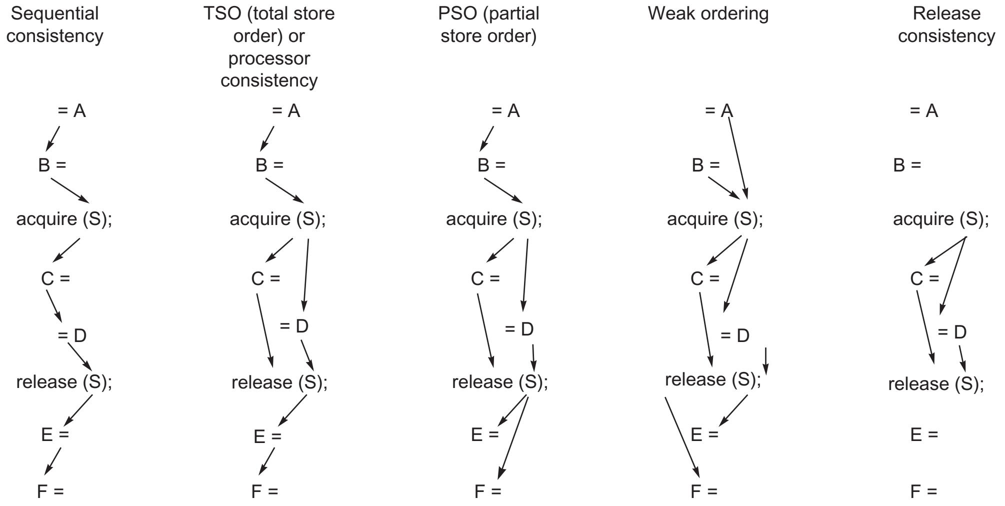


Figure 5.24 These examples of the five consistency models discussed in this section show the reduction in the number of orders imposed as the models become more relaxed. Only the minimum orders are shown with arrows. Orders implied by transitivity, such as the write of C before the release of S in the sequential consistency model or the acquire before the release in weak ordering or release consistency, are not shown.

performance potential from relaxed models, we highly recommend the excellent tutorial by Adve and Gharachorloo (1996).

5.7 Cross-Cutting Issues

Because multiprocessors redefine many system characteristics (e.g., performance assessment, memory latency, and the importance of scalability), they introduce interesting design problems that cut across the spectrum, affecting both hardware and software. In this section, we give several examples related to the issue of memory consistency. We then examine the performance gained when multithreading is added to multiprocessing.

Compiler Optimization and the Consistency Model

Another reason for defining a model for memory consistency is to specify the range of legal compiler optimizations that can be performed on shared data. In explicitly parallel programs, unless the synchronization points are clearly defined and the programs are synchronized, the compiler cannot interchange a read and a write of two different shared data items because such transformations might affect the semantics of the program. This restriction prevents even relatively simple optimizations, such as register allocation of shared data, because such a process usually interchanges reads and writes. In implicitly parallelized programs—for example, those written in High Performance Fortran (HPF)—programs must be synchronized and the synchronization points are known, so this issue does not arise. Whether compilers can get significant advantage from more relaxed consistency models remains an open question, both from a research viewpoint and from a practical viewpoint, where the lack of uniform models is likely to retard progress on deploying compilers.

Using Speculation to Hide Latency in Strict Consistency Models

As we saw in Chapter 3, speculation can be used to hide memory latency. It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory model. The key idea is for the processor to use dynamic scheduling to reorder memory references, letting them possibly execute out of order. Executing the memory references out of order may generate violations of sequential consistency, which might affect the execution of the program. This possibility is avoided by using the delayed commit feature of a speculative processor. Assume the coherency protocol is based on invalidation. If the processor receives an invalidation for a memory reference before the memory reference is committed, the processor uses speculation recovery to back out of the computation and restart with the memory reference whose address was invalidated.

If the reordering of memory requests by the processor yields an execution order that could result in an outcome that differs from what would have been seen under sequential consistency, the processor will redo the execution. The key to using this approach is that the processor need only guarantee that the result would be the same as if all accesses were completed in order, and it can achieve this by detecting when the results might differ. The approach is attractive because the speculative restart will rarely be triggered. It will be triggered only when there are unsynchronized accesses that actually cause a race (Gharachorloo et al., 1992).

Hill (1998) advocated the combination of sequential or processor consistency together with speculative execution as the consistency model of choice. His argument has three parts. First, an aggressive implementation of either sequential consistency or processor consistency will gain most of the advantage of a more relaxed model. Second, such an implementation adds very little to the implementation cost of a speculative processor. Third, such an approach allows the programmer to reason using the simpler programming models of either sequential or processor consistency. The MIPS R10000 design team had this insight in the mid-1990s and used the R10000's out-of-order capability to support this type of aggressive implementation of sequential consistency.

One open question is how successful compiler technology will be in optimizing memory references to shared variables. The state of optimization technology and the fact that shared data are often accessed via pointers or array indexing have limited the use of such optimizations. If this technology were to become available and lead to significant performance advantages, compiler writers would want to be able to take advantage of a more relaxed programming model. This possibility and the desire to keep the future as flexible as possible led the RISC V designers to opt for release consistency, after a long series of debates.

Inclusion and Its Implementation

All multiprocessors use multilevel cache hierarchies to reduce both the demand on the global interconnect and the latency of cache misses. If the cache also provides *multilevel inclusion*—every level of cache hierarchy is a subset of the level farther away from the processor—then we can use the multilevel structure to reduce the contention between coherence traffic and processor traffic that occurs when snoops and processor cache accesses must contend for the cache. Many multiprocessors with multilevel caches enforce the inclusion property, although recent multiprocessors with smaller L1 caches and different block sizes have sometimes chosen not to enforce inclusion. This restriction is also called the *subset property* because each cache is a subset of the cache below it in the hierarchy.

At first glance, preserving the multilevel inclusion property seems trivial. Consider a two-level example: Any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2. Likewise, any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated if it exists.

The catch is what happens when the block sizes of L1 and L2 are different. Choosing different block sizes is quite reasonable, since L2 will be much larger and have a much longer latency component in its miss penalty, and thus will want to use a larger block size. What happens to our "automatic" enforcement of inclusion when the block sizes differ? A block in L2 represents multiple blocks in L1, and a miss in L2 causes the replacement of data that is equivalent to multiple L1 blocks. For example, if the block size of L2 is four times that of L1, then a miss in L2 will replace the equivalent of four L1 blocks. Let's consider a detailed example.

Example Assume that L2 has a block size four times that of L1. Show how a miss for an address that causes a replacement in L1 and L2 can lead to violation of the inclusion property.

Answer Assume that L1 and L2 are direct-mapped and that the block size of L1 is b bytes and the block size of L2 is $4b$ bytes. Suppose L1 contains two blocks with starting addresses x and $x+b$ and that $x \bmod 4b = 0$, meaning that x also is the starting address of a block in L2; then that single block in L2 contains the L1 blocks x, $x+b$, $x+2b$, and $x+3b$. Suppose the processor generates a reference to block y that maps to the block containing x in both caches and thus misses. Because L2 missed, it fetches $4b$ bytes and replaces the block containing x, $x+b$, $x+2b$, and $x+3b$, while L1 takes b bytes and replaces the block containing x. Because L1 still contains $x+b$, but L2 does not, the inclusion property no longer holds.

To maintain inclusion with multiple block sizes, we must probe the higher levels of the hierarchy when a replacement is done at the lower level to ensure that any words replaced in the lower level are invalidated in the higher-level caches; different levels of associativity create the same sort of problems. Baer and Wang (1988) described the advantages and challenges of inclusion in detail, and in 2017 most designers have opted to implement inclusion, often by settling on one block size for all levels in the cache. For example, the Intel i7 uses inclusion for L3, meaning that L3 always includes the contents of all of L2 and L1. This decision allows the i7 to implement a straightforward directory scheme at L3 and to minimize the interference from snooping on L1 and L2 to those circumstances where the directory indicates that L1 or L2 have a cached copy. The AMD Opteron, in contrast, makes L2 inclusive of L1 but has no such restriction for L3. It uses a snooping protocol, but only needs to snoop at L2 unless there is a hit, in which case a snoop is sent to L1.

Performance Gains From Multiprocessing and Multithreading

In this section, we briefly look at a study of the effectiveness of using multithreading on a multicore processor, the IBM Power5; we will return to this topic in the next section, when we examine the performance of the Intel i7. The IBM Power5 is a dual-core that supports simultaneous multithreading (SMT); its basic architecture is very similar to the more recent Power8 (which we examine in the next section), but it has only two cores per processor.

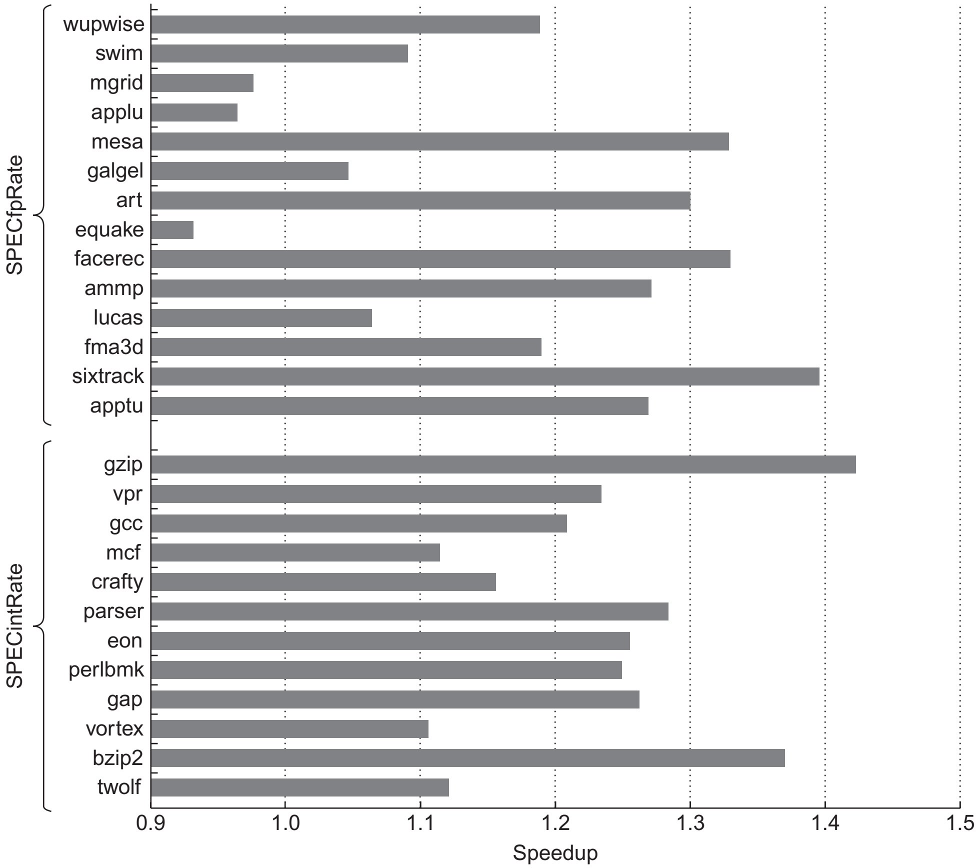


Figure 5.25 A comparison of SMT and single-thread (ST) performance on the 8-processor IBM eServer p5 575 using SPECfpRate (top half) and SPECintRate (bottom half) as benchmarks. Note that the *x*-axis starts at a speedup of 0.9, a performance loss. Only one processor in each Power5 core is active, which should slightly improve the results from SMT by decreasing destructive interference in the memory system. The SMT results are obtained by creating 16 user threads, whereas the ST results use only eight threads; with only one thread per processor, the Power5 is switched to single-threaded mode by the OS. These results were collected by John McCalpin at IBM. As we can see from the data, the standard deviation of the results for the SPECfpRate is higher than for SPECintRate (0.13 versus 0.07), indicating that the SMT improvement for FP programs is likely to vary widely.

To examine the performance of multithreading in a multiprocessor, measurements were made on an IBM system with eight Power5 processors, using only one core on each processor. Figure 5.25 shows the speedup for an 8-processor Power5 multiprocessor, with and without SMT, for the SPECRate2000 benchmarks, as described in the caption. On average, the SPECintRate is 1.23 times faster, and the SPECfpRate is 1.16 times faster. Note that a few floating-point benchmarks experience a slight decrease in performance in SMT mode, with the maximum reduction in speedup being 0.93. Although one might expect that SMT would do a better job of hiding the higher miss rates of the SPECFP benchmarks, it appears that limits in the memory system are encountered when running in SMT mode on such benchmarks.

5.8 Putting It All Together: Multicore Processors and Their Performance

For roughly 10 years, multicore has been the primary focus for scaling performance, although the implementations vary widely, as does their support for larger multichip multiprocessors. In this section, we examine the design of three different multicores, the support they provide for larger multiprocessors, and some performance characteristics, before doing a broader evaluation of small to large multiprocessor Xeon systems, and concluding with a detailed evaluation of the multicore i7 920, a predecessor of the i7 6700.

Performance of Multicore-Based Multiprocessors on a Multiprogrammed Workload

Figure 5.26 shows the key characteristics of three multicore processors designed for server applications and available in 2015 through 2017. The Intel Xeon E7 is based on the same basic design as the i7, but it has more cores, a slightly slower clock rate (power is the limitation), and a larger L3 cache. The Power8 is the newest in the IBM Power series and features more cores and bigger caches. The Fujitsu SPARC64 X+ is the newest SPARC server chip; unlike the T-series mentioned in Chapter 3, it uses SMT. Because these processors are configured for multicore and multiprocessor servers, they are available as a family, varying processor count, cache size, and so on, as the figure shows.

These three systems show a range of techniques both for connecting the on-chip cores and for connecting multiple processor chips. First, let's look at how the cores are connected within a chip. The SPARC64 X+ is the simplest: it shares a single L2 cache, which is 24-way set associative, among the 16 cores. There are four separate DIMM channels to attach memory accessible with a 16 × 4 switch between the cores and the channels.

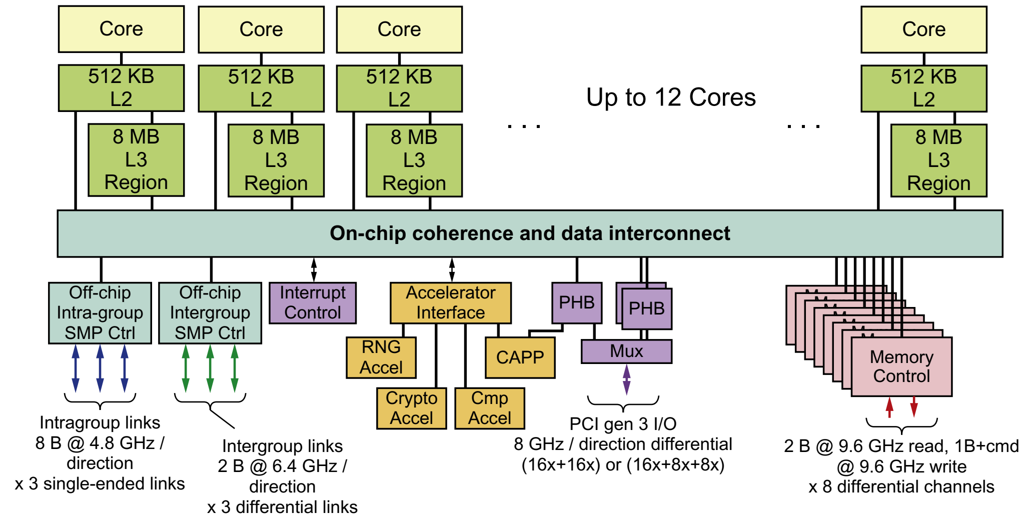
Figure 5.27 shows how the Power8 and Xeon E7 chips are organized. Each core in the Power8 has an 8 MiB bank of L3 directly connected; other banks are accessed via the interconnection network, which has 8 separate buses. Thus the Power8 is a true NUCA (*Nonuniform Cache Architecture*), because the access time to the attached bank of L3 will be much faster than accessing another L3. Each Power8 chip has a set of links that can be used to build a large multiprocessor using an organization we will see shortly. The memory links are connected to a special memory controller that includes an L4 and interfaces directly with DIMMs.

Part B of Figure 5.27, shows how the Xeon E7 processor chip is organized when there are 18 or more cores (20 cores are shown in this figure). Three rings

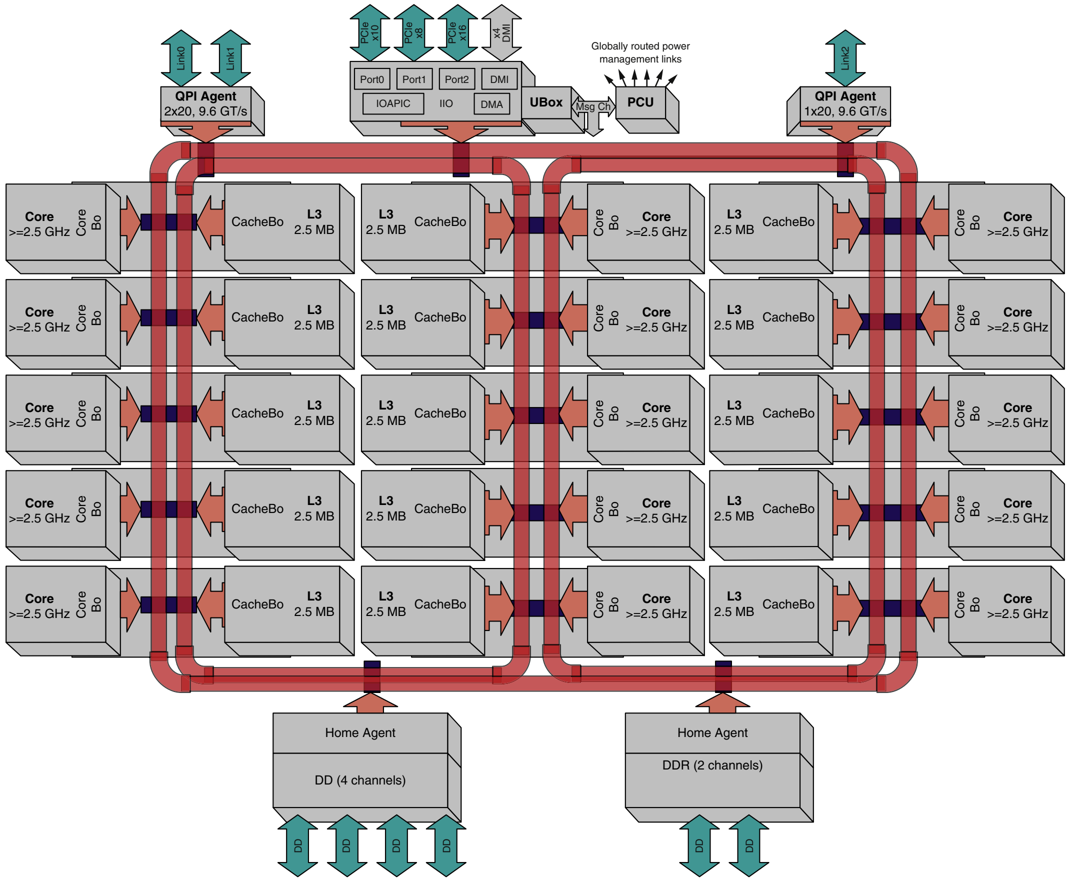
Feature	IBM Power8	Intel Xeon E7	Fujitsu SPARC64 X+
Cores/chip	4, 6, 8, 10, 12	4, 8, 10, 12, 22, 24	16
Multithreading	SMT	SMT	SMT
Threads/core	8	2	2
Clock rate	3.1–3.8 GHz	2.1–3.2 GHz	3.5 GHz
L1 I cache	32 KB per core	32 KB per core	64 KB per core
L1 D cache	64 KB per core	32 KB per core	64 KB per core
L2 cache	512 KB per core	256 KB per core	24 MiB shared
L3 cache	L3: 32–96 MiB: 8 MiB per core (using eDRAM); shared with nonuniform access time	10–60 MiB @ 2.5 MiB per core; shared, with larger core counts	None
Inclusion	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	Extended MESI with behavioral and locality hints (13-states)	MESIF: an extended form of MESI allowing direct transfers of clean blocks	MOESI
Multichip coherence implementation	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory	Hybrid strategy with snooping and directory
Multiprocessor interconnect support	Can connect up to 16 processor chips with 1 or 2 hops to reach any processor	Up to 8 processor chips directly via Quickpath; larger system and directory support with additional logic	Crossbar interconnect chip, supports up to 64 processors; includes directory support
Processor chip range	1–16	2–32	1–64
Core count range	4–192	12–576	8–1024

Figure 5.26 Summary of the characteristics of three recent high-end multicore processors (2015–2017 releases) designed for servers. The table shows the range of processor counts, clock rates, and cache sizes within each processor family. The Power8 L3 is a NUCA (Nonuniform Cache Access) design, and it also supports off-chip L4 of up to 128 MiB using EDRAM. A 32-core Xeon has recently been announced, but no system shipments have occurred. The Fujitsu SPARC64 is also available as an 8-core design, which is normally configured as a single processor system. The last row shows the range of configured systems with published performance data (such as SPECintRate) with both processor chip counts and total core counts. The Xeon systems include multiprocessors that extend the basic interconnect with additional logic; for example, using the standard Quickpath interconnect limits the processor count to 8 and the largest system to $8 \times 24 = 192$ cores, but SGI extends the interconnect (and coherence mechanisms) with extra logic to offer a 32 processor system using 18-core processor chips for a total size of 576 cores. Newer releases of these processors increased clock rates (significantly in the Power8 case, less so in others) and core counts (significantly in the case of Xeon).

connect the cores and the L3 cache banks, and each core and each bank of L3 is connected to two rings. Thus any cache bank or any core is accessible from any other core by choosing the right ring. Therefore, within the chip, the E7 has uniform access time. In practice, however, the E7 is normally operated as a NUMA architecture by logically associating half the cores with each memory channel; this


Core
512 KB L2
8 MB L3 Region
Up to 12 Cores
. . .
On-chip coherence and data interconnect
Off-chip Intra-group SMP Ctrl
Off-chip Intergroup SMP Ctrl
Interrupt Control
Accelerator Interface
RNG Accel
Crypto Accel
Cmp Accel
CAPP
PHB
PHB
Mux
Memory Control
Intragroup links
8 B @ 4.8 GHz / direction
x 3 single-ended links
Intergroup links
2 B @ 6.4 GHz / direction
x 3 differential links
PCI gen 3 I/O
8 GHz / direction differential
(16x+16x) or (16x+8x+8x)
2 B @ 9.6 GHz read, 1B+cmd
@ 9.6 GHz write
x 8 differential channels

(A) The Power8 chip organization


Link0
Link1
QPI Agent
2x20, 9.6 GT/s
PCIe x10
PCIe x8
PCIe x16
x4 DMI
Port0
Port1
Port2
DMI
IOAPIC
IIO
DMA
UBox
Msg Ch
PCU
Globally routed power management links
Link2
QPI Agent
1x20, 9.6 GT/s
Core
>=2.5 GHz
Core Bo
CacheBo
L3
2.5 MB
Home Agent
DD (4 channels)
Home Agent
DDR (2 channels)
DD

(B) The Xeon E7 organization

Figure 5.27 The on-chip organizations of the Power8 and Xeon E7 are shown. The Power8 uses 8 separate buses between L3 and the CPU cores. Each Power8 also has two sets of links for connecting larger multiprocessors. The Xeon uses three rings to connect processors and L3 cache banks, as well QPI for interchip links. Software is used to logically associate half the cores with each memory channel.

increases the probability that a desired memory page is open on a given access. The E7 provides 3 QuickPath Interconnect (QPI) links for connecting multiple E7s.

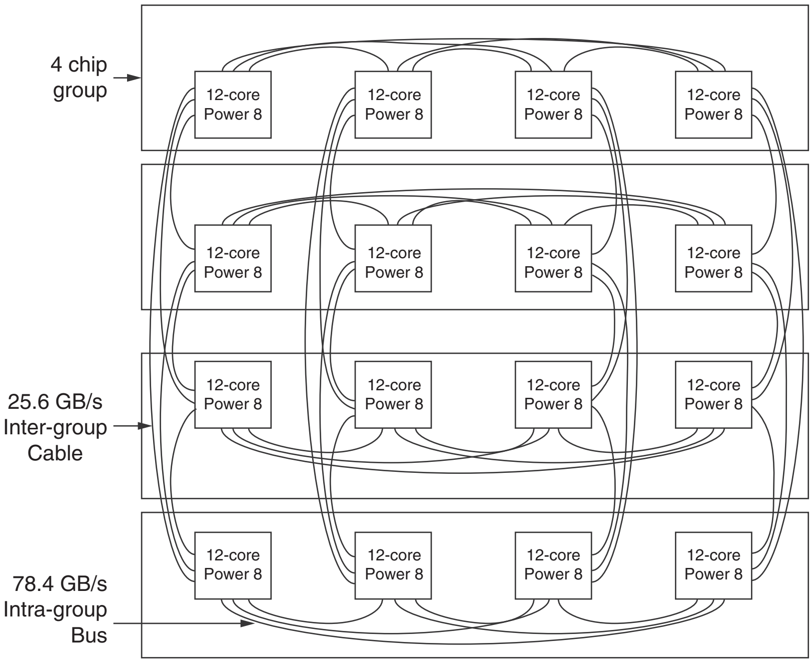
Multiprocessors consisting of these multicores use a variety of different interconnection strategies, as Figure 5.28 shows. The Power8 design provides support for connecting 16 Power8 chips for a total of 192 cores. The intragroup links provide higher bandwidth interconnect among a completely connected module of 4 processor chips. The intergroup links are used to connect each processor chip to the 3 other modules. Thus each processor is two hops from any other, and the memory access time is determined by whether an address resides in local memory, cluster memory, or intercluster memory (actually the latter can have two different values, but the difference is swamped by the intercluster time).

The Xeon E7 uses QPI to interconnect multiple multicore chips. In a 4-chip, multiprocessor, which with the latest announced Xeon could have 128 cores, the three QPI links on each processor are connected to three neighbors, yielding a 4-chip fully connected multiprocessor. Because memory is directly connected to each E7 multicore, even this 4-chip arrangement has nonuniform memory access time (local versus remote). Figure 5.28 shows how 8 E7 processors can be connected; like the Power8, this leads to a situation where every processor is one or two hops from every other processor. There are a number of Xeon-based multiprocessor servers that have more than 8 processor chips. In such designs, the typical organization is to connect 4 processor chips together in a square, as a module, with each processor connecting to two neighbors. The third QPI in each chip is connected to a crossbar switch. Very large systems can be created in this fashion. Memory accesses can then occur at four locations with different timings: local to the processor, an immediate neighbor, the neighbor in the cluster that is two hops away, and through the crossbar. Other organizations are possible and require less than a full crossbar in return for more hops to get to remote memory.

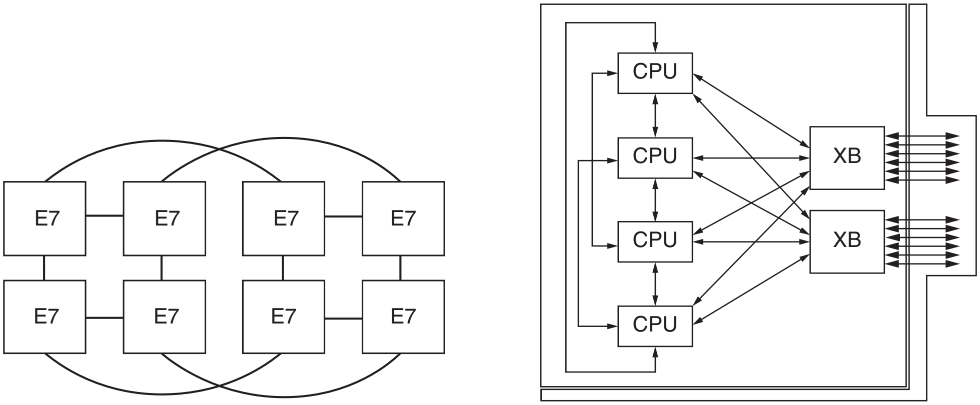
The SPARC64 X+ also uses a 4-processor module, but each processor has three connections to its immediate neighbors plus two (or three in the largest configuration) connections to a crossbar. In the largest configuration, 64 processor chips can be connected to two crossbar switches, for a total of 1024 cores. Memory access is NUMA (local, within a module, and through the crossbar), and coherency is directory-based.

Performance of Multicore-Based Multiprocessors on a Multiprogrammed Workload

First, we compare the performance scalability of these three multicore processors using SPECintRate, considering configurations up to 64 cores. Figure 5.29 shows how the performance scales relative to the performance of the smallest configuration, which varies between 4 and 16 cores. In the plot, the smallest configuration is assumed to have perfect speedup (i.e., 8 for 8 cores, 12 for 12 cores, etc.). This figure does *not* show performance among these different processors. Indeed such performance varies significantly: in the 4-core configuration, the IBM Power8 is


4 chip group
12-core Power 8
25.6 GB/s Inter-group Cable
78.4 GB/s Intra-group Bus

(A) Power8 system with up to 16 chips.


E7
E7
E7
E7
E7
E7
E7
E7
CPU
CPU
CPU
CPU
XB
XB

(B) Xeon E7 system showing with up to 8 chips.

(C) SPARC64 X+ with the 4-chip building block.

Figure 5.28 The system architecture for three multiprocessors built from multicore chips.

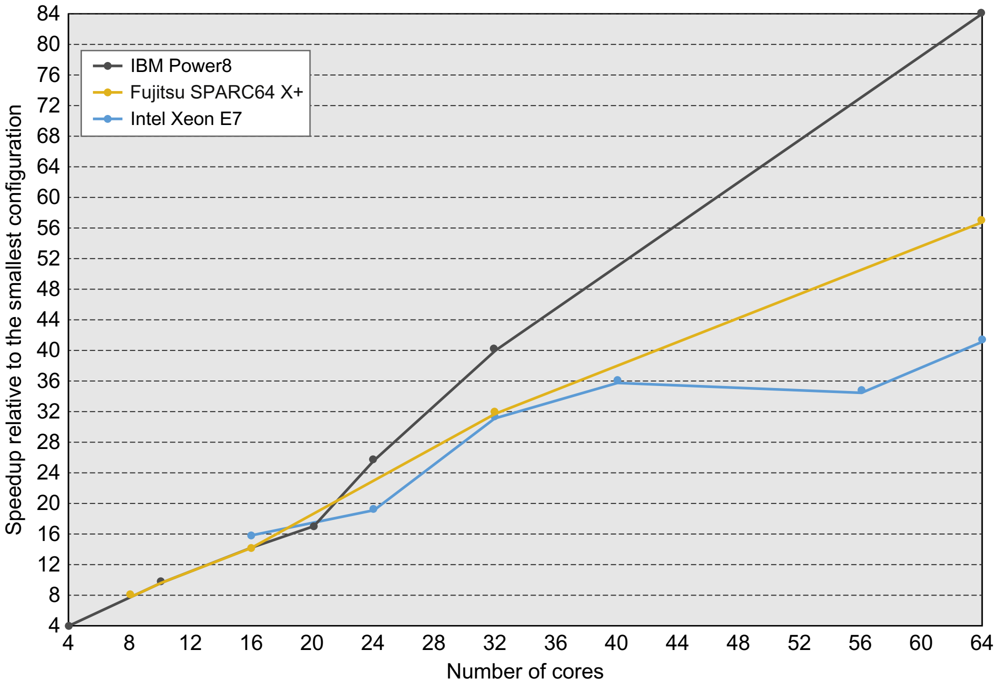


Figure 5.29 The performance scaling on the SPECintRate benchmarks for four multicore processors as the number of cores is increased to 64. Performance for each processor is plotted relative to the smallest configuration and assuming that configuration had perfect speedup. Although this chart shows how a given multiprocessor scales with additional cores, it does not supply any data about performance among processors. There are differences in the clock rates, even within a given processor family. These are generally swamped by the core scaling effects, except for the Power8 that shows a clock range spread of 1.5 × from the smallest configuration to the 64 core configuration.

1.5 times as fast as the SPARC64 X+ on a per core basis! Instead Figure 5.29 shows how the performance scales for each processor family as additional cores are added.

Two of the three processors show diminishing returns as they scale to 64 cores. The Xeon systems appear to show the most degradation at 56 and 64 cores. This may be largely due to having more cores share a smaller L3. For example, the 40-core system uses 4 chips, each with 60 MiB of L3, yielding 6 MiB of L3 per core. The 56-core and 64-core systems also use 4 chips but have 35 or 45 MiB of L3 per chip, or 2.5–2.8 MiB per core. It is likely that the resulting larger L3 miss rates lead to the reduction in speedup for the 56-core and 64-core systems.

The IBM Power8 results are also unusual, appearing to show significant superlinear speedup. This effect, however, is due largely to differences in the clock rates, which are much larger across the Power8 processors than for the other processors

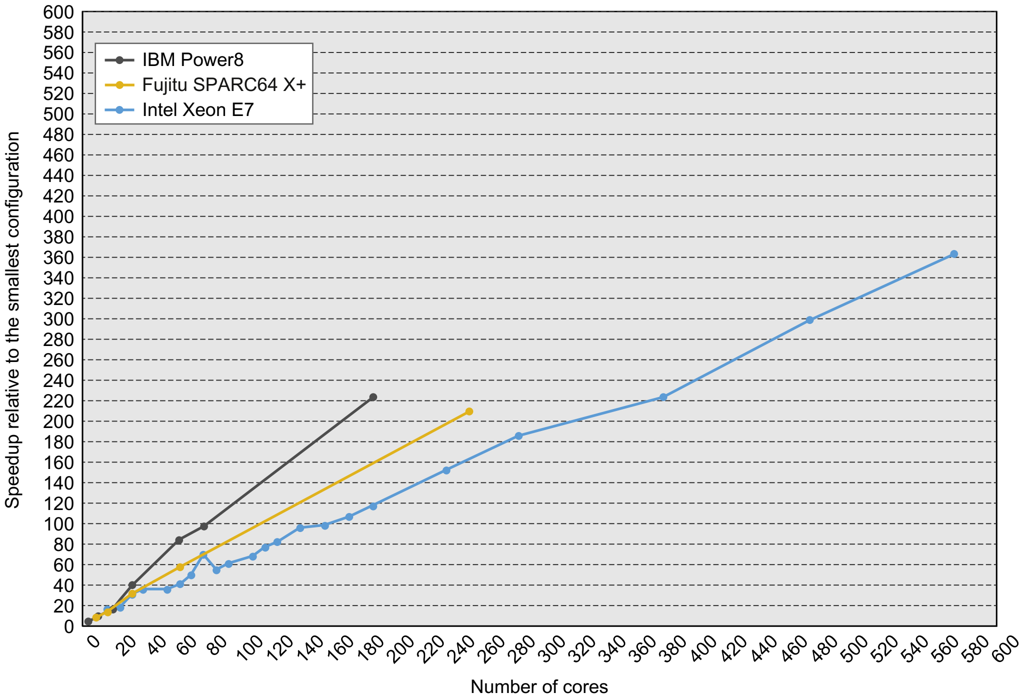


Figure 5.30 The scaling of relative performance for multiprocessor multicore. As before, performance is shown relative to the smallest available system. The Xeon result at 80 cores is the same L3 effect that showed up for smaller configurations. All systems larger than 80 cores have between 2.5 and 3.8 MiB of L3 per core, and the 80-core, or smaller, systems have 6 MiB per core.

in this figure. In particular, the 64-core configuration has the highest clock rate (4.4 GHz), whereas the 4-core configuration has a 3.0 GHz clock. If we normalize the relative speedup for the 64-core system based on the clock rate differential with the 4-core system, the effective speedup is 57 rather than 84. Therefore, while the Power8 system scales well, and perhaps the best among these processors, it is not miraculous.

Figure 5.30 shows scaling for these three systems at configurations above 64 processors. Once again, the clock rate differential explains the Power8 results; the clock-rate equivalent scaled speedup with 192 processors is 167, versus 223, when not accounting for clock rate differences. Even at 167, the Power8 scaling is somewhat better than that on the SPARC64 X+ or Xeon systems. Surprisingly, although there are some effects on speedup in going from the smallest system to 64 cores, they do not seem to get dramatically worse at these larger configurations. The nature of the workload, which is highly parallel and user-CPU-intensive, and the overheads paid in going to 64 cores probably lead to this result.

Scalability in an Xeon MP With Different Workloads

In this section, we focus on the scalability of the Xeon E7 multiprocessors on three different workloads: a Java-based commercially oriented workload, a virtual machine workload, and a scientific parallel processing workload, all from the SPEC benchmarking organization, as described next.

- SPECjbb2015: Models a supermarket IT system that handles a mix of point-of-sale requests, online purchases, and data-mining operations. The performance metric is throughput-oriented, and we use the maximum performance measurement on the server side running multiple Java virtual machines.
- SPECVirt2013: Models a collection of virtual machines running independent mixes of other SPEC benchmarks, including CPU benchmarks, web servers, and mail servers. The system must meet a quality of service guarantee for each virtual machine.
- SPECOMP2012: A collection of 14 scientific and engineering programs written with the OpenMP standard for shared-memory parallel processing. The codes are written in Fortran, C, and C++ and range from fluid dynamics to molecular modeling to image manipulation.

As with the previous results, Figure 5.31 shows performance assuming linear speedup on the smallest configuration, which for these benchmarks varies from 48 cores to 72 cores, and plotting performance relative to the that smallest configuration. The SPECjbb2015 and SPECVirt2013 include significant systems software, including the Java VM software and the VM hypervisor. Other than the system software, the interaction among the processes is very small. In contrast, SPECOMP2012 is a true parallel code with multiple user processes sharing data and collaborating in the computation.

Let's begin by examining SPECjbb2015. It obtains speedup efficiency (speedup/processor ratio) of between 78% and 95%, showing good speedup, even in the largest configuration. SPECVirt2013 does even better (for the range of system measured), obtaining almost linear speedup at 192 cores. Both SPECjbb2015 and SPECVirt2013 are benchmarks that scale up the application size (as in the TPC benchmarks discussed in Chapter 1) with larger systems so that the effects of Amdahl's Law and interprocess communication are minor.

Finally, let's turn to SPECOMP2012, the most compute-intensive of these benchmarks and the one that truly involves parallel processing. The major trend visible here is a steady loss of efficiency as we scale from 30 to 576 cores so that by 576 cores, the system exhibits only half the efficiency it showed at 30 cores. This reduction leads to a relative speedup of 284, assuming that the 30-core speedup is 30. These are probably Amdahl's Law effects resulting from limited parallelism as well as synchronization and communication overheads. Unlike

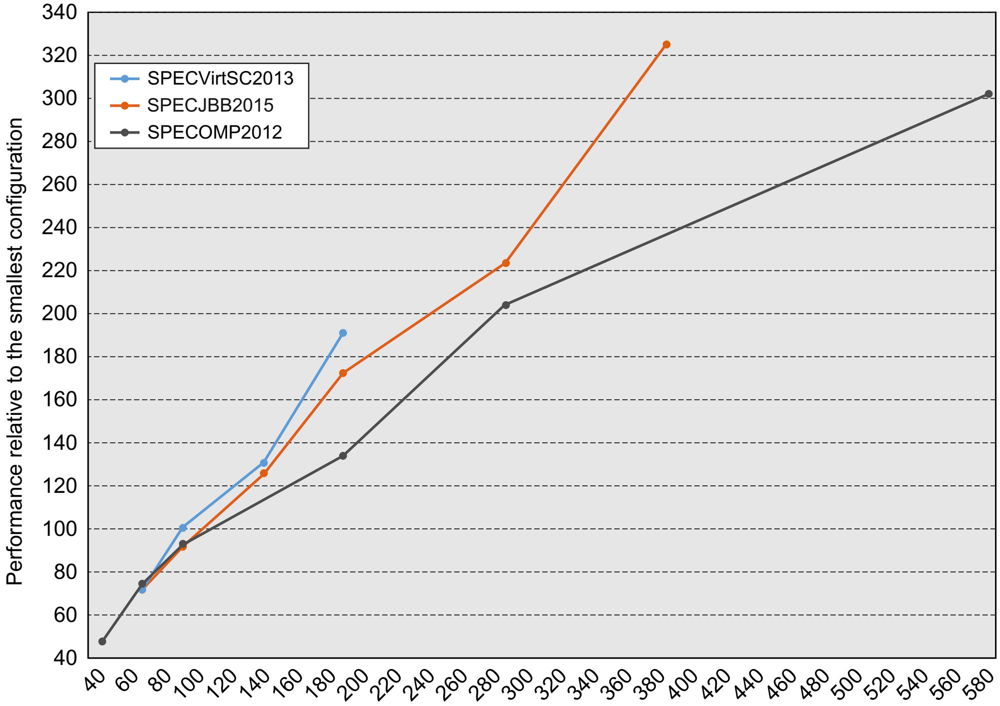


Figure 5.31 Scaling of performance on a range of Xeon E7 systems showing performance relative to the smallest benchmark configuration, and assuming that configuration gets perfect speedup (e.g., the smallest SPEWCOMP configuration is 30 cores and we assume a performance of 30 for that system). Only relative performance can be assessed from this data, and comparisons across the benchmarks have no relevance. Note the difference in the scale of the vertical and horizontal axes.

the SPECjbb2015 and SPECVirt2013, these benchmarks are not scaled for larger systems.

Performance and Energy Efficiency of the Intel i7 920 Multicore

In this section, we closely examine the performance of the i7 920, a predecessor of the 6700, on the same two groups of benchmarks we considered in Chapter 3: the parallel Java benchmarks and the parallel PARSEC benchmarks (described in detail in Figure 3.32 on page 247). Although this study uses the older i7 920, it remains, by far, the most comprehensive study of energy efficiency in multicore processors and the effects of multicore combined with SMT. The fact that the i7 920 and 6700 are similar indicates that the basic insights should also apply to the 6700.

First, we look at the multicore performance and scaling versus a single-core without the use of SMT. Then we combine both the multicore and SMT capability. All the data in this section, like that in the earlier i7 SMT evaluation (Chapter 3) come from Esmaeilzadeh et al. (2011). The dataset is the same as that used earlier (see Figure 3.32 on page 247), except that the Java benchmarks tradebeans and pjbb2005 are removed (leaving only the five scalable Java benchmarks); tradebeans and pjbb2005 never achieve speedup above 1.55 even with four cores and a total of eight threads, and thus are not appropriate for evaluating more cores.

Figure 5.32 plots both the speedup and energy efficiency of the Java and PARSEC benchmarks without the use of SMT. Energy efficiency is computed by the ratio: energy consumed by the single-core run divided by the energy consumed by the two- or four-core run (i.e., efficiency is the inverse of energy consumed). Higher energy efficiency means that the processor consumes less energy for the same computation, with a value of 1.0 being the break-even

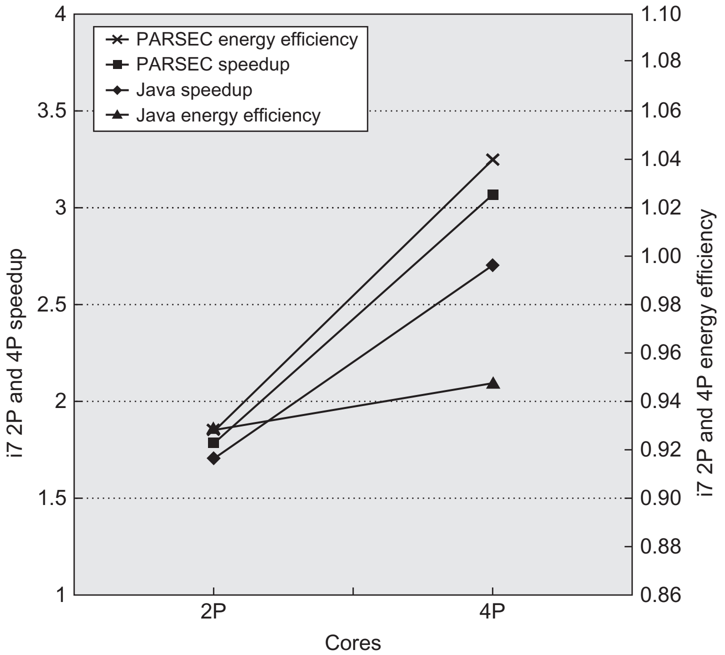


Figure 5.32 This chart shows the speedup and energy efficiency for two- and four-core executions of the parallel Java and PARSEC workloads without SMT. These data were collected by Esmaeilzadeh et al. (2011) using the same setup as described in Chapter 3. Turbo Boost is turned off. The speedup and energy efficiency are summarized using harmonic mean, implying a workload where the total time spent running each benchmark on 2 cores is equivalent.

point. The unused cores in all cases were in deep sleep mode, which minimized their power consumption by essentially turning them off. In comparing the data for the single-core and multicore benchmarks, it is important to remember that the full energy cost of the L3 cache and memory interface is paid in the single-core (as well as the multicore) case. This fact increases the likelihood that energy consumption will improve for applications that scale reasonably well. Harmonic mean is used to summarize results with the implication described in the caption.

As the figure shows, the PARSEC benchmarks get better speedup than the Java benchmarks, achieving 76% speedup efficiency (i.e., actual speedup divided by processor count) on four cores, whereas the Java benchmarks achieve 67% speedup efficiency on four cores. Although this observation is clear from the data, analyzing why this difference exists is difficult. It is quite possible that Amdahl's Law effects have reduced the speedup for the Java workload, which includes some typically serial parts, such as the garbage collector. In addition, interaction between the processor architecture and the application, which affects issues such as the cost of synchronization or communication, may also play a role. In particular, well-parallelized applications, such as those in PARSEC, sometimes benefit from an advantageous ratio between computation and communication, which reduces the dependence on communications costs (see Appendix I).

These differences in speedup translate to differences in energy efficiency. For example, the PARSEC benchmarks actually slightly improve energy efficiency over the single-core version; this result may be significantly affected by the fact that the L3 cache is more effectively used in the multicore runs than in the single-core case and the energy cost is identical in both cases. Thus, for the PARSEC benchmarks, the multicore approach achieves what designers hoped for when they switched from an ILP-focused design to a multicore design; namely, it scales performance as fast or faster than scaling power, resulting in constant or even improved energy efficiency. In the Java case, we see that neither the two- nor four-core runs break even in energy efficiency because of the lower speedup levels of the Java workload (although Java energy efficiency for the 2p run is the same as for PARSEC). The energy efficiency in the four-core Java case is reasonably high (0.94). It is likely that an ILP-centric processor would need *even more* power to achieve a comparable speedup on either the PARSEC or Java workload. Thus the TLP-centric approach is also certainly better than the ILP-centric approach for improving performance for these applications. As we will see in Section 5.10, there are reasons to be pessimistic about simple, efficient, long-term scaling of multicore.

Putting Multicore and SMT Together

Finally, we consider the combination of multicore and multithreading by measuring the two benchmark sets for two to four processors and one to two threads (a total of four data points and up to eight threads). Figure 5.33 shows the speedup

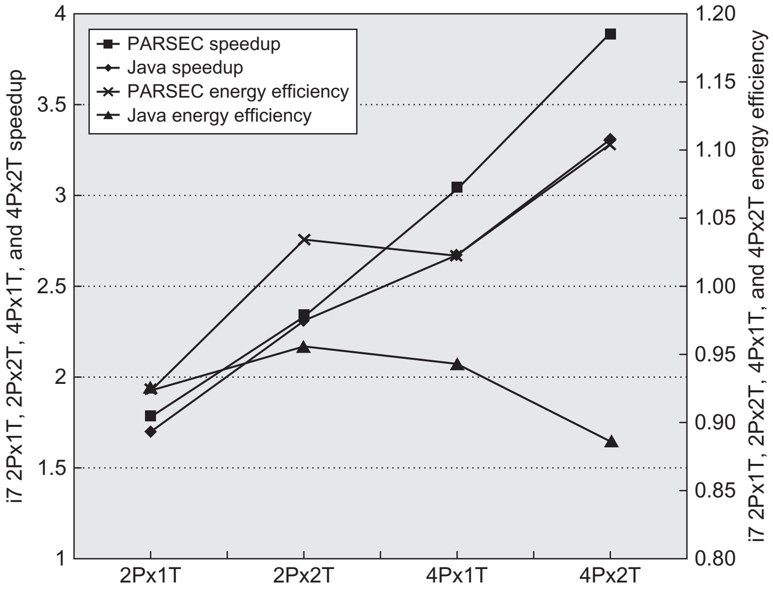


Figure 5.33 This chart shows the speedup for two- and four-core executions of the parallel Java and PARSEC workloads both with and without SMT. Remember that the preceding results vary in the number of threads from two to eight and reflect both architectural effects and application characteristics. Harmonic mean is used to summarize results, as discussed in the Figure 5.32 caption.

and energy efficiency obtained on the Intel i7 when the processor count is two or four and SMT is or is not employed, using harmonic mean to summarize the two benchmarks sets. Clearly, SMT can add to performance when there is sufficient thread-level parallelism available even in the multicore situation. For example, in the four-core, no-SMT case, the speedup efficiencies were 67% and 76% for Java and PARSEC, respectively. With SMT on four cores, those ratios are an astonishing 83% and 97%.

Energy efficiency presents a slightly different picture. In the case of PARSEC, speedup is essentially linear for the four-core SMT case (eight threads), and power scales more slowly, resulting in an energy efficiency of 1.1 for that case. The Java situation is more complex; energy efficiency peaks for the two-core SMT (four-thread) run at 0.97 and drops to 0.89 in the four-core SMT (eight-thread) run. It seems highly likely that the Java benchmarks are encountering Amdahl's Law effects when more than four threads are deployed. As some architects have observed, multicore does shift more responsibility for performance (and thus energy efficiency) to the programmer, and the results for the Java workload certainly bear this out.

5.9 Fallacies and Pitfalls

Given the lack of maturity in our understanding of parallel computing, there are many hidden pitfalls that will be uncovered either by careful designers or by unfortunate ones. Given the large amount of hype that has surrounded multiprocessors over the years, common fallacies abound. We have included a selection of them.

Pitfall *Measuring performance of multiprocessors by linear speedup versus execution time.*

Graphs like those in Figures 5.32 and 5.33, which plot performance versus number of processors, showing linear speedup, a plateau, and then a falling off, have long been used to judge the success of parallel processors. Although speedup is one facet of a parallel program, it is not a direct measure of performance. The first issue is the power of the processors being scaled: a program that linearly improves performance to equal 100 Intel Atom processors (the low-end processor used for netbooks) may be slower than the version run on an 8-core Xeon. Be especially careful of floating-point-intensive programs; processing elements without hardware assist may scale wonderfully but have poor collective performance.

Comparing execution times is fair only if you are comparing the best algorithms on each computer. Comparing the identical code on two computers may seem fair, but it is not; the parallel program may be slower on a uniprocessor than on a sequential version. Developing a parallel program will sometimes lead to algorithmic improvements, so comparing the previously best-known sequential program with the parallel code—which seems fair—will not compare equivalent algorithms. To reflect this issue, the terms *relative speedup* (same program) and *true speedup* (best program) are sometimes used.

Results that suggest *superlinear* performance, when a program on n processors is more than n times faster than the equivalent uniprocessor, may indicate that the comparison is unfair, although there are instances where "real" superlinear speedups have been encountered. For example, some scientific applications regularly achieve superlinear speedup for small increases in processor count (2 or 4 to 8 or 16). These results usually arise because critical data structures that do not fit into the aggregate caches of a multiprocessor with 2 or 4 processors fit into the aggregate cache of a multiprocessor with 8 or 16 processors. As we saw in the previous section, other differences (such as high clock rate) may appear to yield superlinear speedups when comparing slightly different systems.

In summary, comparing performance by comparing speedups is at best tricky and at worst misleading. Comparing the speedups for two different multiprocessors does not necessarily tell us anything about the relative performance of the multiprocessors, as we also saw in the previous section. Even comparing two different algorithms on the same multiprocessor is tricky because we must use true speedup, rather than relative speedup, to obtain a valid comparison.

Fallacy *Amdahl's Law doesn't apply to parallel computers.*

In 1987 the head of a research organization claimed that Amdahl's Law (see Section 1.9) had been broken by an MIMD multiprocessor. This statement hardly meant, however, that the law has been overturned for parallel computers; the neglected portion of the program will still limit performance. To understand the basis of the media reports, let's see what Amdahl (1967) originally said:

> A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. [p. 483]

One interpretation of the law was that, because portions of every program must be sequential, there is a limit to the useful economic number of processors—say, 100. By showing linear speedup with 1000 processors, this interpretation of Amdahl's Law was disproved.

The basis for the statement that Amdahl's Law had been "overcome" was the use of *scaled speedup*, also called *weak scaling*. The researchers scaled the benchmark to have a dataset size that was 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm, the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—thus, linear speedup with 1000 processors. Because the running time grew faster than linear, the program actually ran longer after scaling, even with 1000 processors.

Speedup that assumes scaling of the input is not the same as true speedup, and reporting it as if it were is misleading. Because parallel benchmarks are often run on different-sized multiprocessors, it is important to specify what type of application scaling is permissible and how that scaling should be done. Although simply scaling the data size with processor count is rarely appropriate, assuming a fixed problem size for a much larger processor count (called *strong scaling*) is often inappropriate, as well, because it is likely that users given a much larger multiprocessor would opt to run a larger or more detailed version of an application. See Appendix I for more discussion on this important topic.

Fallacy *Linear speedups are needed to make multiprocessors cost-effective.*

It is widely recognized that one of the major benefits of parallel computing is to offer a "shorter time to solution" than the fastest uniprocessor. Many people, however, also hold the view that parallel processors cannot be as cost-effective as uniprocessors unless they can achieve perfect linear speedup. This argument says that, because the cost of the multiprocessor is a linear function of the number of processors, anything less than linear speedup means that the performance/cost ratio decreases, making a parallel processor less cost-effective than using a uniprocessor.

The problem with this argument is that cost is not only a function of processor count but also depends on memory, I/O, and the overhead of the system (box, power supply, interconnect, etc.). It also makes less sense in the multicore era, when there are multiple processors per chip.

The effect of including memory in the system cost was pointed out by Wood and Hill (1995). We use an example based on more recent data using TPC-C and SPECRate benchmarks, but the argument could also be made with a parallel scientific application workload, which would likely make the case even stronger.

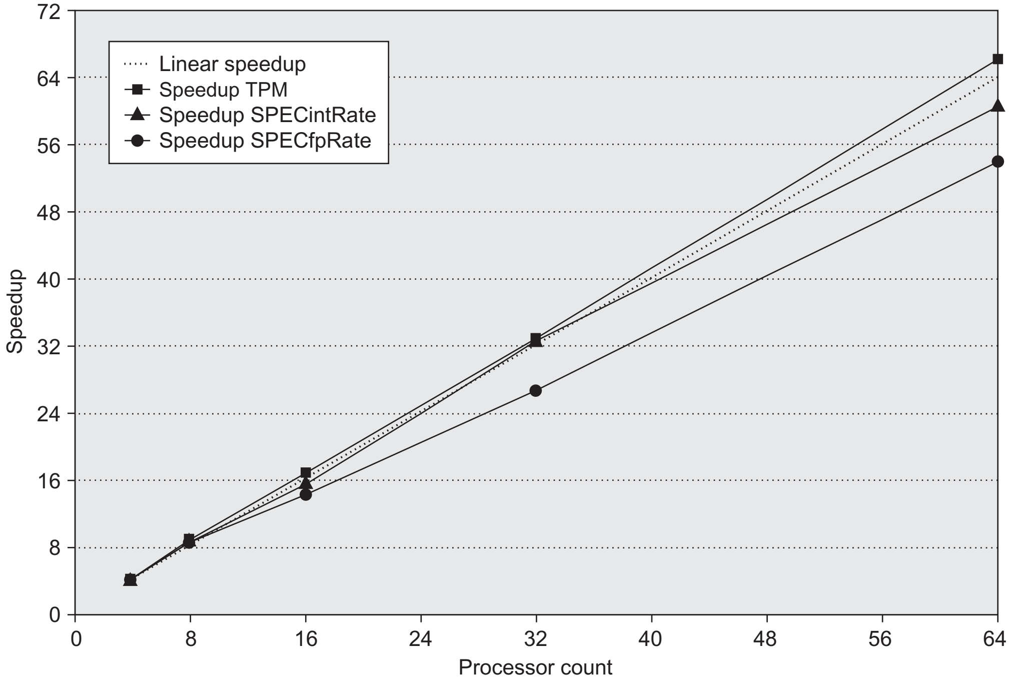


Figure 5.34 Speedup for three benchmarks on an IBM eServer p5 multiprocessor when configured with 4, 8, 16, 32, and 64 processors. The *dashed line* shows linear speedup.

Figure 5.34 shows the speedup for TPC-C, SPECintRate, and SPECfpRate on an IBM eServer p5 multiprocessor configured with 4–64 processors. The figure shows that only TPC-C achieves better than linear speedup. For SPECintRate and SPECfpRate, speedup is less than linear, but so is the cost, because unlike TPC-C, the amount of main memory and disk required both scale less than linearly.

As Figure 5.35 shows, larger processor counts can actually be more cost-effective than the 4-processor configuration. In comparing the cost-performance of two computers, we must be sure to include accurate assessments of both total system cost and what performance is achievable. For many applications with larger memory demands, such a comparison can dramatically increase the attractiveness of using a multiprocessor.

Pitfall *Not developing the software to take advantage of, or optimize for, a multiprocessor architecture.*

There is a long history of software lagging behind on multiprocessors, probably because the software problems are much harder. We give one example to show the subtlety of the issues, but there are many examples we could choose from.

One frequently encountered problem occurs when software designed for a uniprocessor is adapted to a multiprocessor environment. For example, the SGI

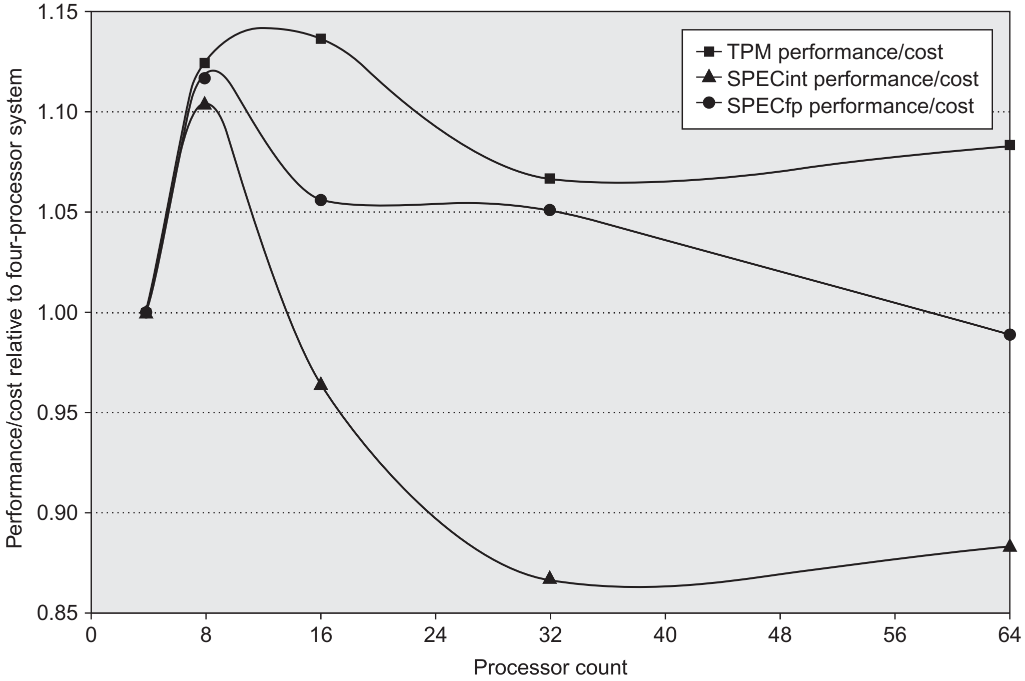


Figure 5.35 The performance/cost for IBM eServer p5 multiprocessors with 4–64 processors is shown relative to the 4-processor configuration. Any measurement above 1.0 indicates that the configuration is more cost-effective than the 4-processor system. The 8-processor configurations show an advantage for all three benchmarks, whereas two of the three benchmarks show a cost-performance advantage in the 16- and 32-processor configurations. For TPC-C, the configurations are those used in the official runs, which means that disk and memory scale nearly linearly with processor count, and a 64-processor machine is approximately twice as expensive as a 32-processor version. In contrast, the disk and memory are scaled more slowly (although still faster than necessary to achieve the best SPECRate at 64 processors). In particular, the disk configurations go from one drive for the 4-processor version to four drives (140 GB) for the 64-processor version. Memory is scaled from 8 GiB for the 4-processor system to 20 GiB for the 64-processor system.

operating system in 2000 originally protected the page table data structure with a single lock, assuming that page allocation was infrequent. In a uniprocessor, this does not represent a performance problem. In a multiprocessor, it can become a major performance bottleneck for some programs.

Consider a program that uses a large number of pages that are initialized at startup, which UNIX does for statically allocated pages. Suppose the program is parallelized so that multiple processes allocate the pages. Because page allocation requires the use of the page table data structure, which is locked whenever it is in use, even an OS kernel that allows multiple threads in the OS will be serialized if the processes all try to allocate their pages at once (which is exactly what we might expect at initialization time).

This page table serialization eliminates parallelism in initialization and has significant impact on overall parallel performance. This performance bottleneck persists even under multiprogramming. For example, suppose we split the parallel program apart into separate processes and run them, one process per processor, so that there is no sharing between the processes. (This is exactly what one user did, because he reasonably believed that the performance problem was due to unintended sharing or interference in his application.) Unfortunately, the lock still serializes all the processes, so even the multiprogramming performance is poor. This pitfall indicates the kind of subtle but significant performance bugs that can arise when software runs on multiprocessors. Like many other key software components, the OS algorithms and data structures must be rethought in a multiprocessor context. Placing locks on smaller portions of the page table effectively eliminates the problem. Similar problems exist in memory structures, which increases the coherence traffic in cases where no sharing is actually occurring.

As multicore became the dominant theme in everything from desktops to servers, the lack of an adequate investment in parallel software became apparent. Given the lack of focus, it will likely be many years before the software systems we use adequately exploit the growing numbers of cores.

5.10 The Future of Multicore Scaling

For more than 30 years, researchers and designers have predicted the end of uniprocessors and their dominance by multiprocessors. Until the early years of this century, this prediction was constantly proven wrong. As we saw in Chapter 3, the costs of trying to find and exploit more ILP became prohibitive in efficiency (both in silicon area and in power). Of course, multicore does not magically solve the power problem because it clearly increases both the transistor count and the active number of transistors switching, which are the two dominant contributions to power. As we will see in this section, energy issues are likely to limit multicore scaling more severely than previously thought.

ILP scaling failed because of both limitations in the ILP available and the efficiency of exploiting that ILP. Similarly, a combination of two factors means that simply scaling performance by adding cores is unlikely to be broadly successful. This combination arises from the challenges posed by Amdahl's Law, which assesses the efficiency of exploiting parallelism, and the end of Dennard's Scaling, which dictates the energy required for a multicore processor.

To understand these factors, we take a simple model of both technology scaling (based on an extensive and highly detailed analysis in Esmaeilzadeh et al. (2012)). Let's start by reviewing energy consumption and power in CMOS. Recall from Chapter 1 that the energy to switch a transistor is given as

$$\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$$

CMOS scaling is limited primarily by thermal power, which is a combination of static leakage power and dynamic power, which tends to dominate. Power is given by

Device count scaling (since a transistor is 1/4 the size)	4
Frequency scaling (based on projections of device speed)	1.75
Voltage scaling projected	0.81
Capacitance scaling projected	0.39
Energy per switched transistor scaling (CV^2)	0.26
Power scaling assuming fraction of transistors switching is the same and chip exhibits full frequency scaling	1.79

Figure 5.36 A comparison of the 22 nm technology of 2016 with a future 11 nm technology, likely to be available sometime between 2022 and 2024. The characteristics of the 11 nm technology are based on the International Technology Roadmap for Semiconductors, which has been recently discontinued because of uncertainty about the continuation of Moore's Law and what scaling characteristics will be seen.

$$\text{Power} = \text{Energy per Transistor} \times \text{Frequency} \times \text{Transistors switched}$$

$$= \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency} \times \text{Transistors switched}$$

To understand the implications of how energy and power scale, let's compare today's 22 nm technology with a technology projected to be available in 2021–24 (depending on the rate at which Moore's Law continues to slow down). Figure 5.36 shows this comparison based on technology projections and resulting effects on energy and power scaling. Notice that power scaling > 1.0 means that the future device consumes more power; in this case, 1.79 × as much.

Consider the implications of this for one of the latest Intel Xeon processors, the E7-8890, which has 24 cores, 7.2 billion transistors (including almost 70 MiB of cache), operates at 2.2 GHz, has a thermal power rating of 165 watts, and a die size of 456 mm^2. The clock frequency is already limited by power dissipation: a 4-core version has a clock of 3.2 GHz, and a 10-core version has a 2.8 GHz clock. With the 11 nm technology, the same size die would accommodate 96 cores with almost 280 MiB of cache and operate at a clock rate (assuming perfect frequency scaling) of 4.9 GHz. Unfortunately, with all cores operating and no efficiency improvements, it would consume $165 \times 1.79 = 295$ watts. If we assume the 165-W heat dissipation limit remains, then only 54 cores can be active. This limit yields a maximum performance speedup of $54/24 = 2.25$ over a 5–6 year period, less than one-half the performance scaling seen in the late 1990s. Furthermore, we may have Amdahl's Law effects, as the next example shows.

Example Suppose we have a 96-core future generation processor, but on average only 54 cores can be busy. Suppose that 90% of the time, we can use all available cores; 9% of the time, we can use 50 cores; and 1% of the time is strictly serial. How much speedup might we expect? Assume that cores can be turned off when not in use and draw no power and assume that the use of a different number of cores is distributed so that we need to worry only about average power consumption. How would the multicore speedup compare to the 24-processor count version that can use all its processor 99% of the time?

Answer We can find how many cores can be used for the 90% of the time when more than 54 are usable, as follows:

$$\text{Average Processor Usage} = 0.09 \times 50 + 0.01 \times 1 + 0.90 \times \text{Max processor}$$
$$54 = 4.51 + 0.90 \times \text{Max processor}$$
$$\text{Max processor} = 55$$

Now, we can find the speedup:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{55}}{55} + \frac{\text{Fraction}_{50}}{50} + (1 - \text{Fraction}_{55} - \text{Fraction}_{50})}$$
$$\text{Speedup} = \frac{1}{\frac{0.90}{55} + \frac{0.09}{50} + 0.01} = 35.5$$

Now compute the speedup on 24 processors:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{24}}{24} + (1 - \text{Fraction}_{24})}$$
$$\text{Speedup} = \frac{1}{\frac{0.99}{24} + 0.01} = 19.5$$

When considering both power constraints and Amdahl's Law effects, the 96-processor version achieves less than a factor of 2 speedup over the 24-processor version. In fact, the speedup from clock rate increase nearly matches the speedup from the 4 × processor count increase. We comment on these issues further in the concluding remarks.

5.11 Concluding Remarks

As we saw in the previous section, multicore does not magically solve the power problem because it clearly increases both the transistor count and the active number of transistors switching, which are the two dominant contributions to power. The failure of Dennard scaling merely makes it more extreme.

But multicore does alter the game. By allowing idle cores to be placed in power-saving mode, some improvement in power efficiency can be achieved, as the results in this chapter have shown. For example, shutting down cores in the Intel i7 allows other cores to operate in Turbo mode. This capability allows a trade-off between higher clock rates with fewer processors and more processors with lower clock rates.

More importantly, multicore shifts the burden for keeping the processor busy by relying more on TLP, which the application and programmer are responsible for identifying, rather than on ILP, for which the hardware is responsible. Multiprogrammed and highly parallel workloads that avoid Amdahl's Law effects will benefit more easily.

Although multicore provides some help with the energy efficiency challenge and shifts much of the burden to the software system, there remain difficult challenges and unresolved questions. For example, attempts to exploit thread-level versions of aggressive speculation have so far met the same fate as their ILP counterparts. That is, the performance gains have been modest and are likely less than the increase in energy consumption, so ideas such as speculative threads or hardware run-ahead have not been successfully incorporated in processors. As in speculation for ILP, unless the speculation is almost always right, the costs exceed the benefits.

Thus, at the present, it seems unlikely that some form of simple multicore scaling will provide a cost-effective path to growing performance. A fundamental problem must be overcome: finding and exploiting significant amounts of parallelism in an energy- and silicon-efficient manner. In the previous chapter, we examined the exploitation of data parallelism via a SIMD approach. In many applications, data parallelism occurs in large amounts, and SIMD is a more energy-efficient method for exploiting data parallelism. In the next chapter, we explore large-scale cloud computing. In such environments, massive amounts of parallelism are available from millions of independent tasks generated by individual users. Amdahl's Law plays little role in limiting the scale of such systems because the tasks (e.g., millions of Google search requests) are independent. Finally, in Chapter 7, we explore the rise of domain-specific architectures (DSAs). Most domain-specific architectures exploit the parallelism of the targeted domain, which is often data parallelism, and as with GPUs, DSAs can achieve much higher efficiency as measured by energy consumption or silicon utilization.

In the last edition, published in 2012, we raised the question of whether it would be worthwhile to consider heterogeneous processors. At that time, no such multicore was delivered or announced, and heterogeneous multiprocessors had seen only limited success in special-purpose computers or embedded systems. While the programming models and software systems remain challenging, it appears inevitable that multiprocessors with heterogeneous processors will play an important role. Combining domain-specific processors, like those discussed in Chapters 4 and 7, with general-purpose processors is perhaps the best road forward to achieve increased performance and energy efficiency while maintaining some of the flexibility that general-purpose processors offer.

5.12 Historical Perspectives and References

Section M.7 (available online) looks at the history of multiprocessors and parallel processing. Divided by both time period and architecture, the section features discussions on early experimental multiprocessors and some of the great debates in parallel processing. Recent advances are also covered. References for further reading are included.

Case Studies and Exercises by Amr Zaky and David A. Wood

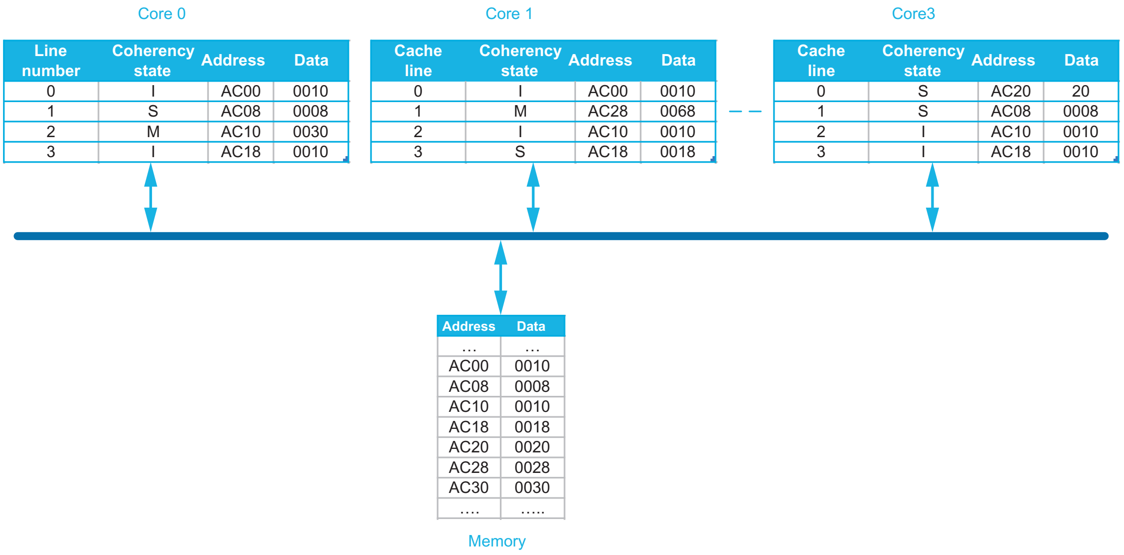
Case Study 1: Single Chip Multicore Multiprocessor

Concepts illustrated by this case study

- Snooping Coherence Protocol Transitions
- Coherence Protocol Performance
- Coherence Protocol Optimizations
- Synchronization

A multicore SMT multiprocessor is illustrated in Figure 5.37. Only the cache contents are shown. Each core has a single, private cache with coherence maintained using the snooping coherence protocol of Figure 5.7. Each cache is direct-mapped, with four lines, each holding 2 bytes (to simplify diagram). For further simplification, the whole line addresses in memory are shown in the address fields in the caches, where the tag would normally exist. The coherence states are denoted M, S, and I for Modified, Shared, and Invalid.

5.1. [10/10/10/10/10/10/10] <5.2> For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form



Core 0

Line number	Coherency state	Address	Data
0	I	AC00	0010
1	S	AC08	0008
2	M	AC10	0030
3	I	AC18	0010

Core 1

Cache line	Coherency state	Address	Data
0	I	AC00	0010
1	M	AC28	0068
2	I	AC10	0010
3	S	AC18	0018

Core3

Cache line	Coherency state	Address	Data
0	S	AC20	20
1	S	AC08	0008
2	I	AC10	0010
3	I	AC18	0010

Memory

Address	Data
…	…
AC00	0010
AC08	0008
AC10	0010
AC18	0018
AC20	0020
AC28	0028
AC30	0030
….	…..

Figure 5.37 Multicore (point-to-point) multiprocessor.

`Ccore#: R, <address>` for reads

and

`Ccore#: W, <address> <-- <value written>` for writes.

For example,

`C3: R, AC10 & C0: W, AC18 <-- 0018`

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:

`C0.L0: (I, AC20, 0001)` indicates that line 0 in core 0 assumes an "invalid" coherence state (`I`), stores `AC20` from the memory, and has data contents `0001`. Furthermore, represent any changes to the memory state as `M: <address> <- value`.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. [10] <5.2> `C0: R, AC20`

b. [10] <5.2> `C0: W, AC20 <-- 80`

c. [10] <5.2> `C3: W, AC20 <-- 80`

d. [10] <5.2> `C1: R, AC10`

e. [10] <5.2> `C0: W, AC08 <-- 48`

f. [10] <5.2> `C0: W, AC30 <-- 78`

g. [10] <5.2> `C3: W, AC30 <-- 78`

5.2. [20/20/20/20] <5.3> The performance of a snooping cache-coherent multiprocessor depends on many detailed implementation issues that determine how quickly a cache responds with data in an exclusive or M state block. In some implementations, a processor read miss to a cache block that is exclusive in another processor's cache is faster than a miss to a block in memory. This is because caches are smaller, and thus faster, than main memory. Conversely, in some implementations, misses satisfied by memory are faster than those satisfied by caches. This is because caches are generally optimized for "front side" or CPU references, rather than "back side" or snooping accesses. For the multiprocessor illustrated in Figure 5.37, consider the execution of a sequence of operations on a single processor core where

- read and write hits generate no stall cycles;
- read and write misses generate N_{memory} and N_{cache} stall cycles if satisfied by memory and cache, respectively;
- write hits that generate an invalidate incur $N_{invalidate}$ stall cycles; and
- a write-back of a block, either due to a conflict or another processor's request to an exclusive block, incurs an additional $N_{writeback}$ stall cycles.

Consider two implementations with different performance characteristics summarized in Figure 5.38.

Parameter	Implementation 1 Cycles	Implementation 2 Cycles
N_{memory}	100	100
N_{cache}	40	130
$N_{invalidate}$	15	15
$N_{writeback}$	10	10

Figure 5.38 Snooping coherence latencies.

To observe how these cycle values are used, we illustrate how the following sequence of operations, assuming the initial caches' states in Figure 5.37, behave under implementation 1.

```
C1: R, AC10
C3: R, AC10
```

For simplicity, assume that the second operation begins after the first completes, even though they are on different processor cores.

For Implementation 1,

- the first read generates 50 stall cycles because the read is satisfied by `C0`'s cache: `C1` stalls for 40 cycles while it waits for the block, and `C0` stalls for 10 cycles while it writes the block back to memory in response to `C1`'s request; and
- the second read by `C3` generates 100 stall cycles because its miss is satisfied by memory.

Therefore this sequence generates a total of 150 stall cycles.

For the following sequences of operations, how many stall cycles are generated by each implementation?

a. [20] <5.3>
```
C0: R, AC20
C0: R, AC28
C0: R, AC30
```

b. [20] <5.3>
```
C0: R, AC00
C0: W, AC08 <-- 48
C0: W, AC30 <-- 78
```

c. [20] <5.3>
```
C1: R, AC20
C1: R, AC28
C1: R, AC30
```

d. [20] <5.3>
```
C1: R, AC00
C1: W, AC08 <-- 48
C1: W, AC30 <-- 78
```

5.3. [20] <5.2> Some applications read a large dataset first and then modify most or all of it. The base MSI coherence protocol will first fetch all of the cache blocks in the Shared state and then be forced to perform an invalidate operation to upgrade them to the

Modified state. The additional delay has a significant impact on some workloads. The MESI addition to the standard protocol (see Section 5.2) provides some relief in these cases. Draw new protocol diagrams for a MESI protocol that adds the Exclusive state and transitions to the base MSI protocol's Modified, Shared, and Invalidate states.

5.4. [20/20/20/20/20] <5.2> Assume the cache contents of Figure 5.37 and the timing of Implementation 1 in Figure 5.38. What are the total stall cycles for the following code sequences with both the base protocol and the new MESI protocol in Exercise 5.3? Assume state transitions that require zero interconnect transactions incur no additional stall cycles.

a. [20] <5.2>
```
C0: R, AC00
C0: W, AC00 <-- 40
```

b. [20] <5.2>
```
C0: R, AC20
C0: W, AC20 <-- 60
```

c. [20] <5.2>
```
C0: R, AC00
C0: R, AC20
```

d. [20] <5.2>
```
C0: R, AC00
C1: W, AC00 <-- 60
```

e. [20] <5.2>
```
C0: R, AC00
C0: W, AC00 <-- 60
C1: W, AC00 <-- 40
```

5.5. Code running on a single core and not sharing any variables with other cores can suffer some performance degradation because of the snooping coherence protocol. Consider the two following iterative loops are NOT functionally equivalent but they seem similar in complexity. One could be led to conclude that they would spend a comparably close number of cycles when executed on the same processor core.

Loop 1	**Loop 2**
`Repeat i: 1 .. n`	`Repeat i:1 .. n`
`A[i] <-- A[i-1] +B[i];`	`A[i] <-- A[i] +B[i];`

Assume that

- every cache line can hold exactly one element of A or B;
- arrays A and B do not interfere in the cache; and
- all the elements of A or B are in the cache before either loop is executed.

Compare their performance when run on a core whose cache uses the **MESI coherence** protocol. Use the stall cycles data for Implementation 1 in Figure 5.38.

Assume that a cache line can hold multiple elements of A and B (A and B go to separate cache lines). How will this affect the relative performances of Loop1 and Loop2?

Suggest hardware and/or software mechanisms that would improve the performance of Loop1 on a single core.

5.6. [20] <5.2> Many snooping coherence protocols have additional states, state transitions, or bus transactions to reduce the overhead of maintaining cache coherency. In Implementation 1 of Exercise 5.2, misses are incurring fewer stall cycles when they are supplied by cache than when they are supplied by memory. The MOESI protocol extension (see Section 5.2) addresses this need.

Draw new protocol diagrams with the additional state and transitions.

5.7. [20/20/20/20] <5.2> For the following code sequences and the timing parameters for the two implementations in Figure 5.36, compute the total stall cycles for the base MSI protocol and the optimized MESI protocol in Exercise 5.3. Assume state transitions that do not require bus transactions incur no additional stall cycles.

a. [20] <5.2>
```
C1: R, AC10
C3: R, AC10
C0: R, AC10
```

b. [20] <5.2>
```
C1: R, AC20
C3: R, AC20
C0: R, AC20
```

c. [20] <5.2>
```
C0: W, AC20 <-- 80
C3: R, AC20
C0: R, AC20
```

d. [20] <5.2>
```
C0: W, AC08 <--88
C3: R, AC08
C0: W, AC08 <-- 98
```

5.8. [20/20/20/20] <5.5> The spin lock is the simplest synchronization mechanism possible on most commercial shared-memory machines. This spin lock relies on the exchange primitive to atomically load the old value and store a new value. The lock routine performs the exchange operation repeatedly until it finds the lock unlocked (i.e., the returned value is `0`).

```
        addi x2, x0, #1
lockit: EXCH x2, 0(x1)
        bnez x2, lockit
```

The lock is released simply by storing a `0` into `x2`.

As discussed in Section 5.5, the more optimized spin lock employs cache coherence and uses a load to check the lock, allowing it to spin with a shared variable in the cache.

```
lockit: ld     x2, 0(x1)
        bnez   x2, lockit
        addi   x2, x0, #1
        EXCH   x2,0(x1)
        bnez   x2, lockit
```

Assume that processor cores C0, C1, and C3 are all trying to acquire a lock at address 0xAC00 (i.e., register R1 holds the value 0xAC00). Assume the cache contents from Figure 5.37 and the timing parameters from Implementation 1 in Figure 5.38. For simplicity, assume the critical sections are 1000 cycles long.

a. [20] <5.5> Using the simple spin lock, determine *approximately* how many memory stall cycles each processor incurs before acquiring the lock.

b. [20] <5.5> Using the optimized spin lock, determine *approximately* how many memory stall cycles each processor incurs before acquiring the lock.

c. [20] <5.5> Using the simple spin lock, *approximately* how many memory accesses occur?

d. [20] <5.5> Using the optimized spin lock, *approximately* how many memory accesses occur?

Case Study 2: Simple Directory-Based Coherence

Concepts illustrated by this case study

- Directory Coherence Protocol Transitions
- Coherence Protocol Performance
- Coherence Protocol Optimizations

Consider the distributed shared-memory system illustrated in Figure 5.39. It consists of 8 nodes of processor cores organized as three-dimensional hypercube with point-to-point interconnections, as shown in the figure. For simplification, we assume the following scaled-down configuration:

- Every node has a *single processor core* with a direct-mapped L1 data cache with its dedicated cache controller.
- The L1 data cache has a capacity of two cache lines with a line size of B bytes.
- The L1 cache states are denoted `M`, `S`, and `I` for Modified, Shared, and Invalid. An example cache entry in some would like

```
1: S, M3, 0xabcd -->
```

Cache line 1 is in the "Shared" state; it contains memory block `M3` and the data value of the block is `0xabcd`.

- The system memory comprises 8 memory blocks (i.e., one memory block per node) and is distributed among the eight nodes, with every node owning a memory block. Node `Ci` owns memory block `Mi`.
- Each memory block is B-bytes wide and is tracked by a coherency directory entry stored with the memory block.

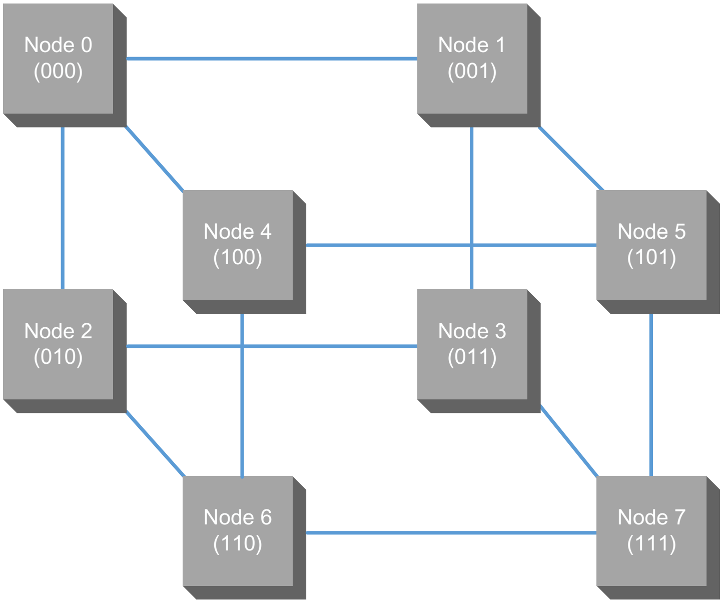

Node 0
(000)
Node 1
(001)
Node 4
(100)
Node 5
(101)
Node 2
(010)
Node 3
(011)
Node 6
(110)
Node 7
(111)

Figure 5.39 Multicore multiprocessor with DSM.

- The state of each memory directory entry is denoted `DM`, `DS`, and `DI` for Directory Modified, Directory Shared, and Directory Invalid. Additionally, the directory entry lists the block sharers using a bit vector with 1 bit for every node. Here is an example memory block and associated directory entry:

  `M3: 0XABCD, DS, 00000011 -->`

  Memory block `M3` (in node C3) contains the value `0xABCD` and is shared by nodes `0` and `1` (corresponding to `1`s in the bit vector).

Read/Write Notation

To describe read/write transactions, we will use the notation

`Ci#: R, <Mi>` for reads

and

`Ci#: W, <Mi> <-- <value written>` for writes.

For example,

`C3: R, M2` describes the core in node 3 issuing a read transaction from an address at memory block `M2` (the address may possibly be cached in `C3` already).

`C0: W, M3 <-- 0018` describes the core in node 0 issuing a write transaction (data is `0X0018`) to an address at memory block `M3` (the address may possibly be cached in `C0` already).

Messages

The directory coherency schemes depend on exchange of command and/or data messages as described by the directory protocol described in Figure 5.20. An example of a command message is a read request. An example of a data message is a read response (with data included).

- Messages originating an ending in the same node do not cross any inter-node links.
- Message with distinct source/destination nodes travel through inter-node links. These messages may be destined from one cache controller to another, from a cache controller to a directory controller, or from a directory controller to a cache controller.
- Messages traveling from a source node to a distinct destination node are statically routed.
    - The static routing algorithm selects a shorted path between the source and destination nodes.
    - The short path is determined by considering the binary representations of the source and destination indices (e.g., `001` for node `C1` and `100` for node `C4`), then by moving from one node to a neighboring node that was not already crossed by the message.
        - For example, to go from node `6` to node `0` (`110 --> 000`), the path is `110--> 100--> 000.`
        - Because more than one shorted path may exist (`110--> 010--> 000` is another path for the preceding example), we assume that the path is selected by inverting first the least significant bit that is different from the corresponding bit in destination index. For example, to travel from node 1 to node 6 (`001--> 110`), the path is `001--> 000--> 010--> 110`.
    - The longest possible path traveled by any message has 3 links (equal to the number of bits in the binary representation of a node index).
- A node can simultaneously process up to three messages from/to distinct neighboring nodes' links if no two of them are competing for the same link resource as clarified by the following examples of **messages sent/received to/from/through** node `000`.

Messages: from `001 --> 010`; `010 --> 000` (to cache/directory controller); `100 --> 001`. **OK** (distinct destinations).

Message: from `001 --> 010`; `000 --> 001` (from cache/directory controller); `100 --> 001`.

Not OK as two messages are destined to node `001`

In case of destination contention, ties are broken assigning priority to

a. message destined to the node (`000` in example) cache or directory controller; then

b. messages forwarded from one to another (through `000` in example); then

c. messages originating from the node (`000` in example) cache or directory controller.

- Assume the transmission and service delays in the following table.

Message type	Cache controller	Directory controller	Link
No data	2 cycles	5 cycles	10 cycles
With data	$(3 + \lceil {}^{B}/_{4} \rceil)$ cycles	$(6 + 10 * B)$ cycles	$(4 + B)$

- If a message is forwarded through a node, it is first completely received by the node before being sent to the next node on the path.
- Assume any cache controller; directory controller has unlimited capacity to enqueue messages and service them in FCFS order.

5.9. [10/10/10] <5.4> For each part of this exercise, assume that initially all caches lines are invalid, and the data in memory `Mi` is the byte `i` (`0X00 <= i <= 0x07`) repeated as many times as the block size. Assume that successive requests are completely serialized. That is, no core will issue a coherency request until the previous request (by same or different core) is completed.

For each of the following parts,

- show the final state (i.e., coherence state, sharers/owners, tags, and data) of the caches and directory controller (including data values) after the given transaction sequence has completed; and
- show the messages transferred (choose a suitable format for message types).

a. [10] <5.4>
```
C3: R, M4
C3: R, M2
C7: W, M4 <--0xaaaa
C1: W, M4 <--0xbbbb
```

b. [10] <5.4>
```
C3: R, M0
C3: R, M2
C6: W, M4 <--0xaaaa
C3: W, M4 <--0xbbbb
```

c. [10] <5.4>
```
C0: R, M7
C3: R, M4
C6: W, M2 <--0xaaaa
C2: W, M2 <--0xbbbb
```

5.10. [10/10/10] <5.4> The directory protocol used in 5.9 (based on Figure 5.20) assumes that the directory controller receives requests, sends invalidates, receives modified data, sends modified data to requester if block was dirty, and so on. Assume now that the directory controller will delegate some work to the cores. For example, it will notify the exclusive owner of a modified block when some other core needs the block and will have the owner send the block to the new sharer. Specifically, consider the following optimizations and indicate what their benefits (if any) are. Also, specify how the messages will be modified (in comparison with Figure 5.20 protocol) to support the new change.

Hint: Benefits might be reduction in number of messages, faster response time, and so on.

a. [10] <5.4> On a write miss to a shared memory block, the directory controller sends the data to the requester and instructs the sharers to send their invalidate acknowledgements directly to the requester.

b. [10] <5.4> On a read miss to a block modified in some other core, the directory controller instructs the owner of the modified copy to directly forward the data to the requester.

c. [10] <5.4> On a read miss to a block in shared (S) state in some other cores, the directory controller instructs one of the sharers (say, the one closest to the requester) to directly forward the data to the requester.

5.11. [15/15/15] <5.4> In problem 5.9, it was assumed that all transactions on the system were serially executed, which is both unrealistic and inefficient in a DSM multicore. We now relax this condition. We will require only that all transactions originating in one core are serialized. However, different cores can independently issue their read/write transactions and even compete for the same memory block. The transactions of problem 5.9 are represented next to reflect the new, relaxed constraints. Redo problem 5.9 with the new, relaxed constraints.

a. [15] <5.4>
```
C1: W, M4 <--0xbbbb    C3: R, M4     C7: R, M2
                       C3: W, M4 <--0xaaaa
```

b. [15] <5.4>
```
C3: R, M0       C6: W, M4 <--0xaaaa
C3: R, M2
C3: W, M4 <--0xbbbb
```

c. [15] <5.4>
```
C0: R, M7  C2: W, M2 <--0xbbbb  C3: R, M4  C6: W, M2 <--0xaaaa
```

5.12. [10/10] <5.4> Use the routing and delay information described earlier and trace how the following groups of transactions will progress in the system (assume that all accesses are misses).

a.
```
C0: R, M7  C2: W, M2 <--0xbbbb  C3: R, M4  C6: W, M2 <--0xaaaa
```

b.
```
C0: R, M7          C3: R, M7
          C2: W, M7 <--0xbbbb
```

5.13. [20] <5.4> What extra complexities may arise if the messages can be adaptively rerouted on the links? For example, a coherency message from core `M1` directory controller to C2 (expressed in binary as $\texttt{M}_{001}$ --> $\texttt{C}_{010}$) will be routed either through the inter-node path $\texttt{C}_{001}$ --> $\texttt{C}_{000}$ --> $\texttt{C}_{010}$ or the inter-node path $\texttt{C}_{001}$ --> $\texttt{C}_{011}$ --> $\texttt{C}_{010}$, depending on link availability.

5.14. [20] <5.4> In a read miss, a cache might overwrite a line in the shared (S) state without notifying the directory that owns the corresponding memory block. Alternatively, it will notify the directory so that it deletes this cache from the list of sharers.

Show how the following transaction groups (performed one at a time in series) will proceed under both approaches.

```
C3:  R, M4
C3:  R, M2
C2:   W, M4 <--0xabcd
```

Case Study 3: Memory Consistency

Concepts Illustrated by This Case Study

- Legitimate Program Behavior Under Sequential Consistency (SC) Models
- Hardware Optimization Allowed for SC Models
- Using Synchronization Primitives to Make a Consistency Model Emulate a More Restrictive Model

5.15. [10/10] <5.6> Consider the following code segments running on two processors P1 and P2. Assume `A` and `B` are initially `0`.

P1:	**P2:**
`While (B == 0);`	`While (A==0);`
`A=1;`	`B = 1;`

a. If the processors adhere to sequential consistency (SC) consistency model. What are the possible values of A and B at the end of the segments? Show the statement interleaving supporting your answer(s).

b. Repeat (a) if the processors adhere to the total store order (TSO) consistency model.

5.16. [5] <5.6> Consider the following code segments running on two processors P1 and P2. Assume `A`, and `B`, are initially `0`. Explain how an optimizing compiler might make it impossible for `B` to be ever set to `2` in a sequentially consistent execution model.

P1:	**P2:**
`A=1;`	`B=1;`
`A=2;`	`While (A <> 1);`
`While (B == 0);`	`B= 2;`

5.17. [10] <5.4>. In a processor implementing a SC consistency model, the data cache is augmented with a data prefetch unit. Will that alter the SC implementation execution results? Why or why not?

5.18. [10/10] <5.6> Assume that the following code segment is executed on a processor that implements partial store order (PSO),

```
A=1;
B=2;
If (C== 3)
D=B;
```

a. Augment the code with synchronization primitives to make it emulate the behavior of a total store order (TSO) implementation.

b. Augment the code with synchronization primitives to make it emulate the behavior of a sequential consistency (SC) implementation.

5.19. [20/20/20] <5.6> Sequential consistency (SC) requires that all reads and writes appear to have executed in some total order. This may require the processor to stall in certain cases before committing a read or write instruction. Consider the code sequence

```
write A
read B
```

where the write `A` results in a cache miss and the read `B` results in a cache hit.

Under SC, the processor must stall read `B` until after it can order (and thus perform) write `A`. Simple implementations of SC will stall the processor until the cache receives the data and can perform the write.

Release consistency (RC) consistency mode (see Section 5.6) relaxes these constraints: ordering—when desired—is enforced by judicious use of synchronization operations. This allows, among other optimizations, processors to implement write buffers, which hold committed writes that have not yet been ordered with respect to other processors' writes. Reads can pass (and potentially bypass) the write buffer in RC (which they could not do in SC).

Assume that one memory operation can be performed per cycle and that operations that hit in the cache or that can be satisfied by the write buffer introduce no stall cycles. Operations that miss incur the latencies listed in Figure 5.38.

How many stall cycles occur *prior* to each operation for both the SC and RC consistency models? (Write buffer can hold at most one write.)

a. [20] <5.6>

```
P0: write 110 <-- 80    //assume miss (no other
                        cache has the line)
P0: read 108            //assume miss (no other
                        cache has the line)
```

b. [20] <5.6>

```
P0: read 110            //assume miss (no other
                        cache has the line)
P0: write 100 <-- 90    //assume hit
```

c. [20] <5.6>

```
P0: write 100 <-- 80    //assume miss
P0: write 110 <-- 90    //assume hit
```

5.20. [20] <5.6> Repeat part (a) of problem 5.19 under an SC model on a processor that has a read prefetch unit. Assume a read prefetch was triggered 20 cycles in advance of the write operation.

Exercises

5.21. [15] <5.1> Assume that we have a function for an application of the form F(i, p), which gives the fraction of time that exactly i processors are usable given that a total of p processors are available. This means that

$$\sum_{i=1}^{p} F(i,p) = 1$$

Assume that when i processors are in use, the applications run i times faster.

a. Rewrite Amdahl's Law so that it gives the speedup as a function of p for some application.

b. An application A runs on single processor for a time T seconds. Different portions of its running time can improve if a larger number of processors is used. Figure 5.40 provides the details.

How much speedup will A achieve when on 8 processors?

c. Repeat for 32 processors and an infinite number of processors.

5.22. [15/20/10] <5.1> In this exercise, we examine the effect of the interconnection network topology on the CPI of programs running on a 64-processor distributed-memory multiprocessor. The processor clock rate is 2.0 GHz, and the base CPI of an application with all references hitting in the cache is 0.75. Assume that 0.2% of the instructions involve a remote communication reference. The cost of a remote communication reference is (100 + 10 h) ns, h being the number of communication network hops that a remote reference has to make to the remote processor memory and back. Assume all communication links are bidirectional.

a. [15] <5.1> Calculate the worst-case remote communication cost when the 64 processors are arranged as a ring, as an 8 × 8 processor grid, or as a hypercube (hint: longest communication path on a 2^n hypercube has n links).

Fraction of T	20%	20%	10%	5%	15%	20%	10%
Processors (P)	1	2	4	6	8	16	128

Figure 5.40 Percentage of application's A time that can use up to P processors.

b. [20] <5.1> Compare the base CPI of the application with no remote communication to the CPI achieved with each of the three topologies in part (a).

5.23. [15] <5.2> Show how the basic snooping protocol of Figure 5.6 can be changed for a write-through cache. What is the major hardware functionality that is not needed with a write-through cache compared with a write-back cache?

5.24. [20/20] <5.2> Please answer the following problems:

a. [20] <5.2> Add a clean exclusive state to the basic snooping cache coherence protocol (Figure 5.6). Show the protocol in the finite state machine format used in the figure.

b. [20] <5.2> Add an "owned" state to the protocol of part (a) and describe using the same finite state machine format used in Figure 5.6.

5.25. [15] <5.2> One proposed solution for the problem of false sharing is to add a valid bit per word. This would allow the protocol to invalidate a word without removing the entire block, letting a processor keep a portion of a block in its cache while another processor writes a different portion of the block. What extra complications are introduced into the basic snooping cache coherence protocol (Figure 5.6) by this addition? Consider all possible protocol actions.

5.26. [15/20] <5.3> This exercise studies the impact of aggressive techniques to exploit instruction-level parallelism in the processor when used in the design of shared-memory multiprocessor systems. Consider two systems identical except for the processor. System A uses a processor with a simple single-issue, in-order pipeline, and system B uses a processor with four-way issue, out-of-order execution and a reorder buffer with 64 entries.

a. [15] <5.3> Following the convention of Figure 5.11, let us divide the execution time into instruction execution, cache access, memory access, and other stalls. How would you expect each of these components to differ between system A and system B?

b. [10] <5.3> Based on the discussion of the behavior of OLTP workload in Section 5.3, what is the important difference between the OLTP workload and other benchmarks that limit benefit from a more aggressive processor design?

5.27. [15] <5.3> How would you change the code of an application to avoid false sharing? What might be done by a compiler and what might require programmer directives?

5.28. [15] <5.3> An application is calculating the number of occurrences of a certain word in a very large number of documents. A very large number of processors divided the work, searching the different documents. They created a huge array—word_count—of 32-bit integers, every element of which is the number of times the word occurred in some document. In a second phase, the computation is moved to a small SMP server with four processors. Each processor sums up approximately ¼ of the array elements. Later, one processor calculates the total sum.

```
for (int p=0; p<=3; p++) // Each iteration of is executed on a
                                    separate processor.
{
     sum [p] = 0;
     for (int i=0; i<n/4; i++) // n is size of word_count and
                                         is divisible by 4
          sum[p] = sum[p] + word_count[p+4*i];
}
total_sum = sum[0] +sum[1]+sum[2]+sum[3] //executed only
                                                    on processor.
```

a. Assuming each processor has a 32-byte L1 data cache. Identify the cache line sharing (true or false) that the code exhibits.

b. Rewrite the code to reduce the number of misses to elements of the array word_count.

c. Identify a manual fix you can make to the code to rid it of any false sharing.

5.29. [15] <5.4> Assume a directory-based cache coherence protocol. The directory currently has information that indicates that processor P1 has the data in "exclusive" mode. If the directory now gets a request for the same cache block from processor P1, what could this mean? What should the directory controller do? (Such cases are called "race conditions" and are the reason why coherence protocols are so hard to design and verify.)

5.30. [20] <5.4> A directory controller can send invalidates for lines that have been replaced by the local cache controller. To avoid such messages, and to keep the directory consistent, replacement hints are used. Such messages tell the controller that a block has been replaced. Modify the directory coherence protocol of Section 5.4 to use such replacement hints.

5.31. [20/15/20/15] <5.4> One downside of a straightforward implementation of directories using fully populated bit vectors is that the total size of the directory information scales as the product: processor count × memory blocks. If memory grows linearly with processor count, the total size of the directory grows quadratically in the processor count. In practice, because the directory needs only 1 bit per memory block (which is typically 32–128 bytes), this problem is not serious for small-to-moderate processor counts. For example, assuming a 128-byte block, and P processors, the amount of directory storage compared to main memory is $P/(128*8) = P/1024$, which is 12.5% overhead for $P = 128$ processors. We can avoid this problem by observing that we need to keep only an amount of information that is proportional to the cache size of each processor. We explore some solutions in these exercises.

a. [20] <5.4> One method to obtain a scalable directory protocol is to organize the multiprocessor as a logical hierarchy with the processors as leaves of the hierarchy and directories positioned at the root of each subtree. The directory at each subtree records which descendants cache which memory blocks. It also records the memory blocks—with a home in that subtree—that are cached outside the subtree. Compute the amount of storage needed to record the processor information for the directories, assuming that each directory is fully associative. Your answer should incorporate both the number of nodes at each level of the hierarchy as well as the total number of nodes.

b. [15] <5.4> Another approach to reducing the directory size is to allow only a limited number of the directory's memory blocks to be shared at any given time. Implement the directory as a four-way set-associative cache storing full bit vectors. If a directory cache miss occurs, choose a directory entry and invalidate the entry. Describe how this organization will work elaborating what will happen as a is block read, written replaced and written back to memory. Modify the protocol in Figure 5.20 to reflect the new transitions required by this directory organization.

c. [20] <5.4> Rather than reducing the number of directory entries, we can implement bit vectors that are not dense. For example, we can set every directory entry to 9 bits. If a block is cached in only one node outside its home, this field contains the node number. If the block is cached in more than one node outside its home, this field is a bit vector with each bit indicating a group of eight processors, at least one of which caches the block. Illustrate how this scheme would work for a 64-processor DSM machine that consists of eight 8-processors groups.

d. [15] An extreme approach to reducing the directory size is to implement an "empty" directory; that is, the directory in every processor does not store any memory states. It receives requests and forwards them as *appropriate*. What is the benefit of having such a directory over having no directory at all for a DSM system?

5.32. [10] <5.5> Implement the classical compare-and-swap instruction using the *load linked/store conditional* instruction pair.

5.33. [15] <5.5> One performance optimization commonly used is to pad synchronization variables so as not to have any other useful data in the same cache line. Construct an example demonstrating that this optimization can be extremely useful in some situations. Assume a snoopy write invalidate protocol.

5.34. [30] <5.5> One possible implementation of the *load linked/store conditional pair* for multicore processors is to constrain these instructions to using uncached memory operations. A monitor unit intercepts all reads and writes from any core to the memory. It keeps track of the source of the *load linked* instructions and whether any intervening stores occur between the *load linked* and its corresponding *store conditional* instruction. The monitor can prevent any failing store conditional from writing any data and can use the interconnect signals to inform the processor that this store failed.

Design such a monitor for a memory system supporting a four-core SMP. Take into account that, generally, read and write requests can have different data sizes (4/8/16/32 bytes). Any memory location can be the target of a *load linked/store* *conditional* pair, and the memory monitor should assume that *load linked/store conditional* references to any location can, possibly, be interleaved with regular accesses to the same location. The monitor complexity should be independent of the memory size.

5.35. [25] <5.5> Prove that, in a two-level cache hierarchy where L1 is closer to the processor, inclusion is maintained with no extra action if L2 has at least as much associativity as L1, both caches use LRU replacement, and both caches have the same block sizes.

5.36. [Discussion] <5> When trying to perform detailed performance evaluation of a multiprocessor system, system designers use one of three tools: analytical models, trace-driven simulation, and execution-driven simulation. Analytical models use mathematical expressions to model the behavior of programs. Trace-driven simulations run the applications on a real machine and generate a trace, typically of memory operations. These traces can be replayed through a cache simulator or a simulator with a simple processor model to predict the performance of the system when various parameters are changed. Execution-driven simulators simulate the entire execution maintaining an equivalent structure for the processor state and so on.

a. What are the accuracy/speed trade-offs between these approaches?

b. CPU traces, if not carefully collected, can exhibit artifacts of the system they are collected on. Discuss this issue while using branch-prediction and spin-wait synchronization as examples. (Hint: The program itself is not available to a pure CPU trace; just the trace is available.)

5.37. [40] <5.7, 5.9> Multiprocessors and clusters usually show performance increases as you increase the number of the processors, with the ideal being n times speedup for n processors. The goal of this biased benchmark is to make a program that gets worse performance as you add processors. For example, this means that one processor on the multiprocessor or cluster runs the program fastest, two are slower, four are slower than two, and so on. What are the key performance characteristics for each organization that give inverse linear speedup?

This page intentionally left blank

6.1 Introduction 466
6.2 Programming Models and Workloads for Warehouse-Scale Computers 471
6.3 Computer Architecture of Warehouse-Scale Computers 477
6.4 The Efficiency and Cost of Warehouse-Scale Computers 482
6.5 Cloud Computing: The Return of Utility Computing 490
6.6 Cross-Cutting Issues 501
6.7 Putting It All Together: A Google Warehouse-Scale Computer 503
6.8 Fallacies and Pitfalls 514
6.9 Concluding Remarks 518
6.10 Historical Perspectives and References 519
Case Studies and Exercises by Parthasarathy Ranganathan 519

6

Warehouse-Scale Computers to Exploit Request-Level and Data-Level Parallelism

The datacenter is the computer.

Luiz André Barroso,
Google (2007)

A hundred years ago, companies stopped generating their own power with steam engines and dynamos and plugged into the newly built electric grid. The cheap power pumped out by electric utilities didn't just change how businesses operate. It set off a chain reaction of economic and social transformations that brought the modern world into existence. Today, a similar revolution is under way. Hooked up to the Internet's global computing grid, massive information-processing plants have begun pumping data and software code into our homes and businesses. This time, it's computing that's turning into a utility.

Nicholas Carr,
The Big Switch: Rewiring the World, from Edison to Google (2008)

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00006-7
© 2019 Elsevier Inc. All rights reserved.

6.1 Introduction

> Anyone can build a fast CPU. The trick is to build a fast system.
>
> **Seymour Cray,**
> *Considered the father of the supercomputer*

The warehouse-scale computer (WSC)[1] is the foundation of Internet services billions of people use every day: search, social networking, online maps, video sharing, online shopping, email services, and so on. The tremendous popularity of such Internet services necessitated the creation of WSCs that could keep up with the rapid demands of the public. Although WSCs may appear to be just large data centers, their architecture and operation are quite different, as we will see. Today's WSCs act as one giant machine that costs hundreds of million dollars for the building, the electrical and cooling infrastructure, the servers, and the networking equipment that connects and houses 50,000–100,000 servers. Moreover, the rapid growth of commercial cloud computing (see Section 6.5) makes WSCs accessible to anyone with a credit card.

Computer architecture extends naturally to designing WSCs. For example, Luiz Barroso of Google (quoted earlier) did his dissertation research in computer architecture. He believes that an architect's skills of designing for scale, designing for dependability, and a knack for debugging hardware are very helpful in the creation and operation of WSCs.

At this leading-edge scale, which requires innovation in power distribution, cooling, monitoring, and operations, the WSC is the modern descendant of the supercomputer—making Seymour Cray the godfather of today's WSC architects. His extreme computers handled computations that could be done nowhere else, but were so expensive that only a few companies could afford them. This time the target is providing information technology for the world instead of high-performance computing (HPC) for scientists and engineers; thus WSCs arguably play a more important role for society today than Cray's supercomputers did in the past.

Unquestionably, WSCs have many orders of magnitude more users than high-performance computing, and they represent a much greater share of the IT market. Whether measured by the number of users or revenue, Google is 1000 times larger than Cray Research ever was.

---

[1] This chapter is based on material from the book *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, by Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle of Google (2013); the blog Perspectives at mvdirona.com and the talks "Cloud-Computing Economies of Scale" and "Data Center Networks Are in My Way," by James Hamilton of Amazon Web Services (2009, 2010); and the paper *Above the Clouds: A View of Cloud Computing*, by Michael Armbrust et al. (2010).

WSC architects share many goals and requirements with server architects:

- *Cost-performance*—Work done per dollar is critical in part because of the scale. Reducing the costs of a collection of WSCs by few percent could save millions of dollars.
- *Energy efficiency*—Except for the photons that leave WSCs, they are essentially closed systems, with almost all the energy consumed turned into heat that must be removed. Thus, peak power and consumed power drive both the cost of power distribution and the cost of cooling systems. The majority of the infrastructure costs of building a WSC goes toward power and cooling. Moreover, energy efficiency is an important part of environmental stewardship. Therefore, work done per joule is critical for both WSCs and its servers because of the high cost of building the power and mechanical infrastructure for a warehouse of computers and for the resulting monthly utility bills.
- *Dependability via redundancy*—The long-running nature of Internet services means that the hardware and software in a WSC must collectively provide at least 99.99% (called "four nines") of availability; that is, services must be down less than 1 h per year. Redundancy is the key to dependability for both WSCs and servers. Although server architects often utilize more hardware at higher costs to reach high availability, WSC architects rely instead on numerous cost-effective servers connected by a network and redundancy managed by software. In addition to local redundancy inside a WSC, an organization needs redundant WSCs to mask events that can take out whole WSCs. Indeed, although every cloud service needs to be available at least 99.99% of the time, the dependability of a full Internet company like Amazon, Google, or Microsoft needs to be even higher. If one of these companies was completely offline for 1 h per year—that is, 99.99% availability—that would be front page news. Multiple WSCs have the added benefit of reducing latency for services that are widely deployed (Figures 6.18–6.20).
- *Network I/O*—Server architects must provide a good network interface to the external world, and WSC architects must also. Networking is needed to keep data consistent between multiple WSCs as well as to interface with the public.
- *Both interactive and batch processing workloads*—Although one expects highly interactive workloads for services like search and social networking with billions of users, WSCs, like servers, also run massively parallel batch programs to calculate metadata useful to such services. For example, MapReduce jobs are run to convert the pages returned from crawling the web into search indices (see Section 6.2).

Not surprisingly, there are also characteristics *not* shared with server architecture:

- *Ample parallelism*—A concern for a server architect is whether the applications in the targeted marketplace have enough concurrency to justify the amount of parallel hardware and whether the cost is too high for sufficient communication hardware to exploit this parallelism. A WSC architect has no such concern. First, batch applications benefit from the large number of distinct datasets that require independent processing, such as billions of web pages from a web crawl. This processing is *data-level parallelism*, which we saw in Chapter 4, this time applied to data in storage instead of data in memory. Second, interactive Internet service applications, also known as *software as a service* (*SaaS*), can benefit from millions of independent users of interactive Internet services. Reads and writes are seldom dependent in SaaS, so SaaS rarely needs to synchronize. For example, search uses a read-only index and email normally reads and writes independent information. We call this type of easy parallelism *request-level parallelism*, as many independent efforts can proceed in parallel naturally with little need for communication or synchronization; an example is that journal-based updating can reduce throughput demands. Even read-/write-dependent features are sometimes dropped to offer storage that can scale to the size of modern WSCs. In any case, WSC applications have no choice but to find algorithms that can scale across hundreds to thousands of servers, as that is what customers expect and that is what the WSC technology provides.

- *Operational costs count*—Server architects typically ignore operational costs of a server, assuming that they pale in comparison to purchase costs. WSCs have longer lifetimes—the building and electrical and cooling infrastructure are often amortized 10–15 years—so the operational costs add up: energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years.
- *Location counts*—To build a WSC, the first step is building a warehouse. One question is where? Real estate agents emphasize location, but the location for a WSC means access to water, inexpensive electricity, proximity to Internet backbone optical fibers, people nearby to work in the WSC, and low risk from environmental disasters, such as earthquakes, floods, and hurricanes. A more obvious concern is just the cost of the land, including enough space to grow the WSC. For companies with many WSCs, another concern is finding a place geographically near a current or future population of Internet users, to reduce latency over the Internet. Other factors include taxes, property costs, social issues (people sometimes want a facility in their country), political issues (some jurisdictions require local hosting), cost of networking, reliability of networking, cost of power, source of power (e.g., hydroelectric versus coal), weather (cooler is cheaper, as Section 6.4 shows), and overall Internet connectivity (Australia is close to Singapore geographically, but the network link bandwidth between them is not great).
- *Computing efficiently at low utilization*—Server architects usually design their systems for peak performance within a cost budget and worry about power only to make sure they don't exceed the cooling capacity of their enclosure. As we will see (Figure 6.3), WSC servers are rarely fully utilized, in part to ensure low response time and in part to offer the redundancy needed to deliver dependable computing. Given that operational costs count, such servers need to compute efficiently at all utilization levels.
- *Scale and the opportunities/problems associated with scale*—Often extreme computers are extremely expensive because they require custom hardware,

and yet the cost of customization cannot be effectively amortized since few extreme computers are made. However, when purchasing thousands of servers at a time, there *are* great volume discounts. WSCs are so massive internally that there is economy of scale even if there are not many WSCs. As we will see in Sections 6.5 and 6.10, these economies of scale led to commercial cloud computing because the lower per-unit costs of a WSC meant that companies could rent servers at a profit below what it costs outsiders to do so themselves. The flip side of 100,000 servers is failures. Figure 6.1 shows outages and anomalies for 2400 servers. Even if a server had a mean time to failure (MTTF) of an amazing 25 years (200,000 h), the WSC architect would need to design for five server failures a day. Figure 6.1 lists the annualized disk failure rate as 2%–10%. Given two disks per server and an annual failure rate of 4%, with 100,000 servers the WSC architect should expect to see one disk fail per *hour*. However, software failures vastly outnumber hardware failures, as Figure 6.1 shows, so the system design must be resilient to server crashes caused by software bugs, which would happen even more frequently than disk failures. With the thousands of servers in these very large facilities, WSC operators become very good at changing disks, so the cost of disk failure is much lower for a WSC than a small data center. The same applies to DRAMs. Plausibly, WSCs could use even less reliable components if cheaper ones were available.

Approx. number events in 1st year	Cause	Consequence
1 or 2	Power utility failures	Lose power to whole WSC; doesn't bring down WSC if UPS and generators work (generators work about 99% of time).
4	Cluster upgrades	Planned outage to upgrade infrastructure, many times for evolving networking needs such as recabling, to switch firmware upgrades, and so on. There are about nine planned cluster outages for every unplanned outage.
1000s	Hard-drive failures	2%–10% annual disk failure rate (Pinheiro et al., 2007)
	Slow disks	Still operate, but run 10× to 20× more slowly
	Bad memories	One uncorrectable DRAM error per year (Schroeder et al., 2009)
	Misconfigured machines	Configuration led to ~30% of service disruptions (Barroso and HÖlzle, 2009)
	Flaky machines	1% of servers reboot more than once a week (Barroso and HÖlzle, 2009)
5000	Individual server crashes	Machine reboot; typically takes about 5 min (caused by problems in software or hardware).

Figure 6.1 List of outages and anomalies with the approximate frequencies of occurrences in the first year of a new cluster of 2400 servers. We label what Google calls a cluster an *array*; see Figure 6.5. Based on Barroso, L.A., 2010. Warehouse Scale Computing [keynote address]. In: Proceedings of ACM SIGMOD, June 8–10, 2010, Indianapolis, IN.

Example Calculate the availability of a service running on the 2400 servers in Figure 6.1. Unlike a service in a real WSC, in this example the service cannot tolerate hardware or software failures. Assume that the time to reboot software is 5 min and the time to repair hardware is 1 h.

Answer We can estimate service availability by calculating the time of outages because of failures of each component. We'll conservatively take the lowest number in each category in Figure 6.1 and split the 1000 outages evenly between four components. We ignore slow disks—the fifth component of the 1000 outages—because they hurt performance but not availability, and power utility failures, because the uninterruptible power supply (UPS) system hides 99% of them.

$$\begin{aligned}\text{Hours Outage}_{\text{service}} &= (4+250+250+250)\times 1\,\text{h} + (250+5000)\times 5\,\text{min}\\ &= 754+438 = 1192\,\text{h}\end{aligned}$$

Since there are 365 × 24 or 8760 h in a year, availability is

$$\text{Availability}_{\text{system}} = \left(\frac{8760-1192}{8760}\right) = \frac{7568}{8760} = 86\%$$

Without software redundancy to mask the many outages, a service on those 2400 servers would be down on average one day a week—zero "nines"—which is far below the 99.99% of availability is the goal of WSCs.

As Section 6.10 explains, the forerunners of WSCs are *computer clusters*. Clusters are collections of independent computers that are connected together using local area networks (LANs) and switches. For workloads that did not require intensive communication, clusters offered much more cost-effective computing than shared-memory multiprocessors. (Shared-memory multiprocessors were the forerunners of the multicore computers discussed in Chapter 5.) Clusters became popular in the late 1990s for scientific computing and then later for Internet services. One view of WSCs is that they are just the logical evolution from clusters of hundreds of servers to tens of thousands of servers.

A natural question is whether WSCs are similar to modern clusters for high-performance computing. Although some have similar scale and cost—there are HPC designs with a million processors that cost hundreds of millions of dollars—they historically have had more powerful processors and much lower-latency networks between the nodes than are found in WSCs because the HPC applications are more interdependent and communicate more frequently (see Section 6.3). The programming environment also emphasizes thread-level parallelism or data-level parallelism (see Chapters 4 and 5), typically emphasizing latency to complete a single task in contrast to bandwidth to complete many independent tasks via request-level parallelism. The HPC clusters also tend to have long-running jobs that keep the servers fully utilized, even for weeks at a time, whereas the utilization of servers in WSCs ranges between 10% and 50% (see Figure 6.3 on page 441) and varies every day. Unlike supercomputer environments, thousands of developers work on the WSC code base and deploy significant software releases every week (Barroso et al., 2017).

How do WSCs compare to conventional data centers? The operators of a traditional data center generally collect machines and third-party software from many parts of an organization and run them centrally for others. Their main focus tends to be consolidation of the many services onto fewer machines, which are isolated from each other to protect sensitive information. Thus, virtual machines are increasingly important in data centers. Virtual machines are important for WSCs as well, but they play a different role. They are used to offer isolation between different customers and to slice hardware resources into different-sized shares to rent at several price points (see Section 6.5). Unlike WSCs, conventional data centers tend to have a great deal of hardware and software heterogeneity to serve their varied customers inside an organization. WSC programmers customize third-party software or build their own, and WSCs have much more homogeneous hardware; the WSC goal is to make the hardware/software in the warehouse act like a single computer that typically runs a variety of applications. Often the biggest cost in a conventional data center is the people to maintain it, whereas, as we will see in Section 6.4, in a well-designed WSC, the server hardware is the greatest cost, and people costs shift from the topmost to the bottommost. Conventional data centers also don't have the scale of a WSC, so they don't get the economic benefits of the scale previously mentioned.

Thus, although a WSC might be considered as an extreme data center in that computers are housed separately in a space with special electrical and cooling infrastructure, traditional data centers share little with the challenges and opportunities of a WSC, either architecturally or operationally.

We start the introduction to WSCs with their workload and a programming model.

6.2 Programming Models and Workloads for Warehouse-Scale Computers

If a problem has no solution, it may not be a problem, but a fact—not to be solved, but to be coped with over time.

Shimon Peres

In addition to the public-facing Internet services such as search, video sharing, and social networking that make them famous, WSCs also run batch applications, such as converting videos into new formats or creating search indexes from web crawls.

A popular framework for batch processing in a WSC is MapReduce (Dean and Ghemawat, 2008) and its open-source twin Hadoop. Figure 6.2 shows the increasing popularity of MapReduce at Google over time. Inspired by the Lisp functions

Month	Number of MapReduce Jobs	Average completion time (s)	Average no. servers per job	Avg. no. cores per server	CPU core years	Input data (PB)	Intermediate data (PB)	Output data (PB)
Sep-16	95,775,891	331	130	2.4	311,691	11,553	4095	6982
Sep-15	115,375,750	231	120	2.7	272,322	8307	3980	5801
Sep-14	55,913,646	412	142	1.9	200,778	5989	2530	3951
Sep-13	28,328,775	469	137	1.4	81,992	2579	1193	1684
Sep-12	15,662,118	480	142	1.8	60,987	2171	818	874
Sep-11	7,961,481	499	147	2.2	40,993	1162	276	333
Sep-10	5,207,069	714	164	1.6	30,262	573	139	37
Sep-09	4,114,919	515	156	3.2	33,582	548	118	99
Sep-07	2,217,000	395	394	1.0	11,081	394	34	14
Mar-06	171,000	874	268	1.6	2002	51	7	3
Aug-04	29,000	634	157	1.9	217	3.2	0.7	0.2

Figure 6.2 Monthly MapReduce usage at Google from 2004 to 2016. Over 12 years the number of MapReduce jobs increased by a factor of 3300. Figure 6.17 on page 461 estimates that running the September 2016 workload on Amazon's cloud computing service EC2 would cost \$114 million. Updated from Dean, J., 2009. Designs, lessons and advice from building large distributed systems [keynote address]. In: Proceedings of 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware, Co-located with the 22nd ACM Symposium on Operating Systems Principles, October 11–14, 2009, Big Sky, Mont.

of the same name, Map first applies a programmer-supplied function to each logical input record. Map runs on hundreds of computers to produce an intermediate result of key-value pairs. Reduce collects the output of those distributed tasks and collapses them using another programmer-defined function. Assuming the Reduce function is commutative and associative, it can run in log N time. With appropriate software support, both functions are fast yet easy to understand and use. Within 30 min, a novice programmer can run a MapReduce task on thousands of computers.

Figure 6.2 shows the average job uses hundreds of servers. Other than a few highly tuned applications from high-performance computing, such MapReduce jobs are the most parallel applications today, whether measured in total CPU time or number of servers utilized.

Here is a MapReduce program that calculates the number of occurrences of every English word in a large collection of documents. Following is a simplified version of that program, which shows just the inner loop and that assumes only one occurrence of all English words found in a document (Dean and Ghemawat, 2008):

```
map(String key, String value):
        // key: document name
        // value: document contents
        for each word w in value:
        EmitIntermediate(w, "1"); // Produce list of
        all words

reduce(String key, Iterator values):
        // key: a word
        // values: a list of counts
        int result = 0;
        for each v in values:

        result += ParseInt(v); // get integer from key-
        value pair
        Emit(AsString(result));
```

The function `EmitIntermediate` used in the Map function emits each word in the document and the value one. Then the Reduce function sums all the values per word for each document using ParseInt() to get the number of occurrences per word in all documents. The MapReduce runtime environment schedules map tasks and reduce tasks to the nodes of a WSC. (The complete version of the program is found in Dean and Ghemawat (2008).)

MapReduce can be thought of as a generalization of the single instruction stream, multiple data streams (SIMD) operation (Chapter 4)—except that a function to be applied is passed to the data—that is followed by a function that is used in a reduction of the output from the Map task. Because reductions are commonplace even in SIMD programs, SIMD hardware often offers special operations for the reductions. For example, Intel's AVX SIMD instructions include "horizontal" instructions that add pairs of operands that are adjacent in registers.

To accommodate variability in performance from hundreds of computers, the MapReduce scheduler assigns new tasks based on how quickly nodes complete prior tasks. Obviously, a single slow task can hold up completion of a large MapReduce job. Dean and Barroso (2013) label such a situation *tail latency*. In a WSC, the solution to slow tasks is to provide software mechanisms to cope with such variability that is inherent at this scale. This approach is in sharp contrast to the solution for a server in a conventional data center, where traditionally slow tasks mean hardware is broken and needs to be replaced or that server software needs tuning and rewriting. Performance heterogeneity is the norm for 50,000–100,000 servers in a WSC. For example, toward the end of a MapReduce program, the system will start backup executions on other nodes of the tasks that haven't completed yet and take the result from whichever finishes first. In return for increasing resource usage a few percentage points, Dean and Ghemawat (2008) found that some large tasks completed 30% faster.

Dependability was built into MapReduce from the start. For example, each node in a MapReduce job is required to report back to the master node periodically with a list of completed tasks and with updated status. If a node does not report back by the deadline, the master node deems the node dead and reassigns the node's work to other nodes. Given the amount of equipment in a WSC, it's not surprising that failures are commonplace, as the prior example attests. To deliver on 99.99% availability, systems software must cope with this reality in a WSC. To reduce operational costs, all WSCs use automated monitoring software allowing one operator to be responsible for more than 1000 servers.

Programming frameworks such as MapReduce for batch processing and externally facing SaaS such as Search rely upon internal software services for their success. For example, MapReduce relies on the Google File System (GFS) (Ghemawat et al., 2003) or on Colossus (Fikes, 2010) to supply files to any computer, so that MapReduce tasks can be scheduled anywhere.

In addition to GFS and Colossus, examples of these scalable storage systems include Amazon's key value storage system Dynamo (DeCandia et al., 2007) and the Google record storage system BigTable (Chang et al., 2006). Note that such systems often build upon each other. For example, BigTable stores its logs and data on GFS or Colossus, much as a relational database may use the file system provided by the kernel operating system.

These internal services usually make different decisions than similar software running on single servers. For example, rather than assuming storage is reliable, such as by using RAID storage servers, these systems often make complete replicas of the data. Replicas can help with read performance as well as with availability; with proper placement, replicas can overcome many other system failures, like those in Figure 6.1. Systems like Colossus use error-correcting codes rather than full replicas to reduce storage costs, but the constant is cross-server redundancy rather than within-a-server or within-a-storage array redundancy. Thus, failure of the entire server or storage device doesn't negatively affect availability of the data.

Another example of the different approach is that WSC storage software often uses relaxed consistency rather than following all the ACID (atomicity, consistency, isolation, and durability) requirements of conventional database systems. The insight is that it's important for multiple replicas of data to agree *some time*, but, for most applications, they do not need to be in agreement at all times. For example, eventual consistency is fine for video sharing. Eventual consistency makes storage systems much easier to scale, which is an absolute requirement for WSCs.

The workload demands of these public interactive services all vary considerably; even a prominent global service such as Google Search varies by a factor of two depending on the time of day. When factoring in weekends, holidays, and popular times of year for some applications—such as photograph-sharing services after New Year's Day or online shopping before Christmas—a much greater variation in server utilization becomes apparent. Figure 6.3 shows average utilization of 5000 Google servers over a 6-month period. Note that less than 0.5% of servers averaged 100% utilization, and most servers operated between 10% and 50% utilization. Stated alternatively, just 10% of all servers were utilized more than 50%. Thus, it's much more important for servers in a WSC to perform

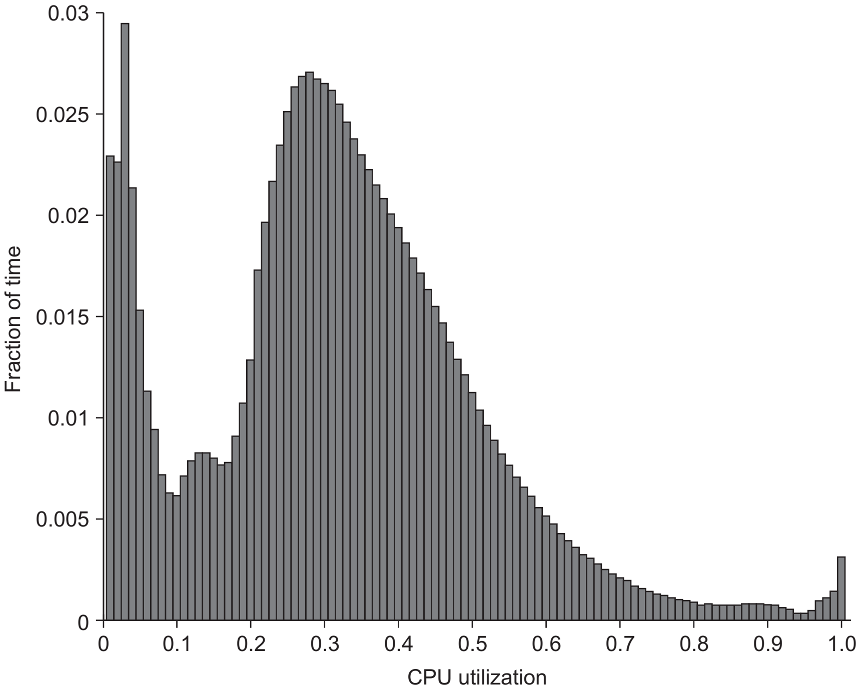


Figure 6.3 Average CPU utilization of more than 5000 servers during a 6-month period at Google. Servers are rarely completely idle or fully utilized, instead operating most of the time at between 10% and 50% of their maximum utilization. The third column from the right in Figure 6.4 calculates percentages plus or minus 5% to come up with the weightings; thus 1.2% for the 90% row means that 1.2% of servers were between 85% and 95% utilized. From Figure 1 in Barroso, L.A., Hölzle, U., 2007. The case for energy-proportional computing. IEEE Comput. 40 (12), 33–37.

well while doing little than to perform efficiently only at their peak, as they rarely operate at their peak.

In summary, WSC hardware and software must cope with variability in load based on user demand and in performance and dependability because of the vagaries of hardware at this scale.

Example As a result of measurements like those in Figure 6.3, the SPECpower benchmark measures power and performance from 0% load to 100% in 10% increments (see Chapter 1). The overall single metric that summarizes this benchmark is the sum of all the performance measures (server-side Java operations per second) divided by the sum of all power measurements in watts. Thus, each level is assumed to be equally likely. How would the numbers summary metric change if the levels were weighted by the utilization frequencies in Figure 6.3?

Load	Performance	Watts	SPEC weightings	Weighted performance	Weighted watts	Figure 6.3 weightings	Weighted performance	Weighted watts
100%	2,889,020	662	9.09%	262,638	60	0.80%	22,206	5
90%	2,611,130	617	9.09%	237,375	56	1.20%	31,756	8
80%	2,319,900	576	9.09%	210,900	52	1.50%	35,889	9
70%	2,031,260	533	9.09%	184,660	48	2.10%	42,491	11
60%	1,740,980	490	9.09%	158,271	45	5.10%	88,082	25
50%	1,448,810	451	9.09%	131,710	41	11.50%	166,335	52
40%	1,159,760	416	9.09%	105,433	38	19.10%	221,165	79
30%	869,077	382	9.09%	79,007	35	24.60%	213,929	94
20%	581,126	351	9.09%	52,830	32	15.30%	88,769	54
10%	290,762	308	9.09%	26,433	28	8.00%	23,198	25
0%	0	181	9.09%	0	16	10.90%	0	20
Total	15,941,825	4967		1,449,257	452		933,820	380
				ssj_ops/W	3210		ssj_ops/W	2454

Figure 6.4 SPECpower result using the weightings from Figure 6.3 instead of even weightings.

Answer Figure 6.4 shows the original weightings and the new weighting that match Figure 6.3. These weightings reduce the performance summary by 30% from 3210 ssj_ops/watt to 2454.

Given the scale, software must handle failures, which means there is little reason to buy “gold-plated” hardware that reduces the frequency of failures. The primary impact would be to increase cost. Barroso and Hölzle (2009) found a factor of 20 difference in price-performance between a high-end Hewlett Packard shared-memory multiprocessor and a commodity Hewlett Packard server when running the TPC-C database benchmark. Not surprisingly, Google and all other companies with WSCs use low-end commodity servers. In fact, the Open Compute Project (http://opencompute.org) is an organization where such companies collaborate on open designs of servers and racks for data centers.

Such WSC services also tend to develop their own software rather than buy third-party commercial software, in part to cope with the huge scale and in part to save money. For example, even on the best price-performance platform for TPC-C in 2017, adding the cost of the SAP SQL Anywhere database and the Windows operating system increases the cost of the Dell PowerEdge T620 server by 40%. In contrast, Google runs BigTable and the Linux operating system on its servers, for which it pays no licensing fees.

Given this review of the applications and systems software of a WSC, we are ready to look at the computer architecture of a WSC.

6.3 Computer Architecture of Warehouse-Scale Computers

Networks are the connective tissue that binds 50,000–100,000 servers together. Analogous to the memory hierarchy of Chapter 2, WSCs use a hierarchy of networks. Figure 6.5 shows one example. Ideally, the combined network would provide nearly the performance of a custom high-end switch for 100,000 servers at about the cost per port of a commodity switch designed for 50 servers. As we will see in Section 6.6, networks for WSCs are an area of active innovation.

The structure that holds the servers is a rack. Although the width of racks varies per WSC—some are the classic 19-in. wide rack; others are two or three times wider—the height tends to be no higher than 6–7 ft since people must service them. Such a rack has roughly 40–80 servers. Because it is often convenient to connect the network cables at the top of the rack, this switch is commonly called a *Top of Rack* (*ToR*) switch. (Some WSCs have racks with multiple ToR switches.) Typically, the bandwidth within the rack is much higher than between racks,

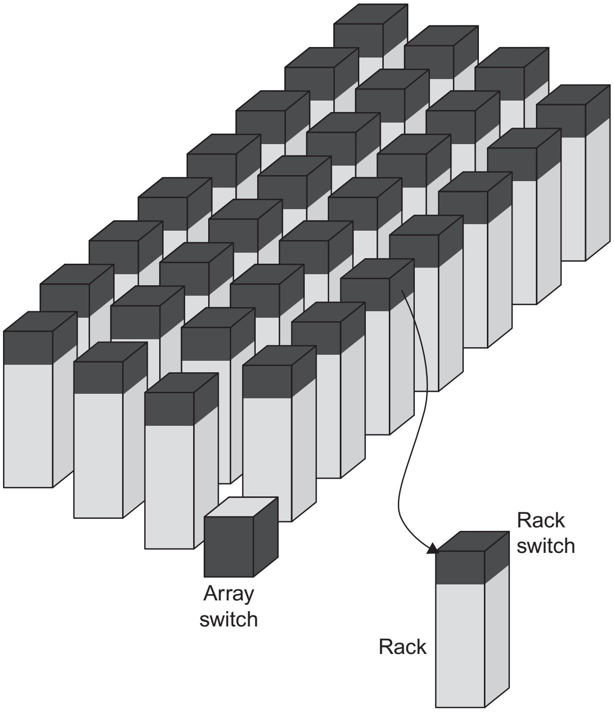


Figure 6.5 Hierarchy of switches in a WSC. Based on Figure 1.1 in Barroso, L.A., Clidaras, J., Hölzle, U., 2013. The datacenter as a computer: an introduction to the design of warehouse-scale machines. Synth. Lect. Comput. Architect. 8 (3), 1–154.

so it matters less where the software places the sender and the receiver if they are within the same rack. This flexibility is ideal from a software perspective.

These switches often offer 4–16 uplinks, which leave the rack to go to the next higher switch in the network hierarchy. Thus, the bandwidth leaving the rack is 6–24 times smaller than the bandwidth within the rack. This ratio is called *oversubscription*. However, large oversubscription means programmers must be aware of the performance consequences when placing senders and receivers in different racks. This increased software-scheduling burden is another argument for network switches designed specifically for the data center.

The switch that connects an array of racks is considerably more expensive than the ToR switch. This cost is due in part because of the higher connectivity and in part because the bandwidth through the switch must be much greater to reduce the oversubscription problem. Barroso et al. (2013) reported that a switch having 10 times the *bisection bandwidth*—basically, the worst-case internal bandwidth—of a rack switch costs about 100 times as much. One reason is that the cost of switch bandwidth for n ports can grow as n^2. Sections 6.6 and 6.7 describe the networking above the ToR switch in great detail.

Storage

A natural design is to fill a rack with servers, minus whatever space needed for the switches. This design leaves open the question of where the storage is placed. From a hardware construction perspective, the simplest solution would be to include disks inside the rack and rely on Ethernet connectivity for access to information on the disks of remote servers. An expensive alternative would be to use network-attached storage (NAS), perhaps over a storage network like InfiniBand. In the past, WSCs generally relied on local disks and provided storage software that handled connectivity and dependability. For example, GFS used local disks and maintained replicas to overcome dependability problems. This redundancy covered not only local disk failures but also power failures to racks and to whole clusters. The flexibility of GFS's eventual consistency lowers the cost of keeping replicas consistent, which also reduces the network bandwidth requirements of the storage system.

Today the storage options are considerably more varied. Although some racks are balanced in terms of servers and disks, as in the past, there may also be racks deployed without local disks and some racks loaded with disks. System software today often uses RAID-like error correction codes to lower the storage cost of dependability.

Be aware that there is confusion about the term *cluster* when talking about the architecture of a WSC. Using the definition in Section 6.1, a WSC is just an extremely large cluster. In contrast, Barroso et al. (2013) used the term cluster to mean the next-sized grouping of computers, containing many racks. In this chapter, to avoid confusion, we will use the term *array* to mean a large collection of racks organized in rows, preserving the original definition of the word cluster to represent anything from a collection of networked computers within a rack to an entire warehouse full of networked computers.

WSC Memory Hierarchy

Figure 6.6 shows the latency, bandwidth, and capacity of memory hierarchy inside a WSC, and Figure 6.7 shows the same data visually. These figures are based on the following assumptions (Barroso et al., 2013):

	Local	Rack	Array
DRAM latency (μs)	0.1	300	500
Flash latency (μs)	100	400	600
Disk latency (μs)	10,000	11,000	12,000
DRAM bandwidth (MB/s)	20,000	100	10
Flash bandwidth (MB/s)	1000	100	10
Disk bandwidth (MB/s)	200	100	10
DRAM capacity (GB)	16	1024	31,200
Flash capacity (GB)	128	20,000	600,000
Disk capacity (GB)	2000	160,000	4,800,000

Figure 6.6 Latency, bandwidth, and capacity of the memory hierarchy of a WSC (Barroso et al., 2013). Figure 6.7 plots this same information.

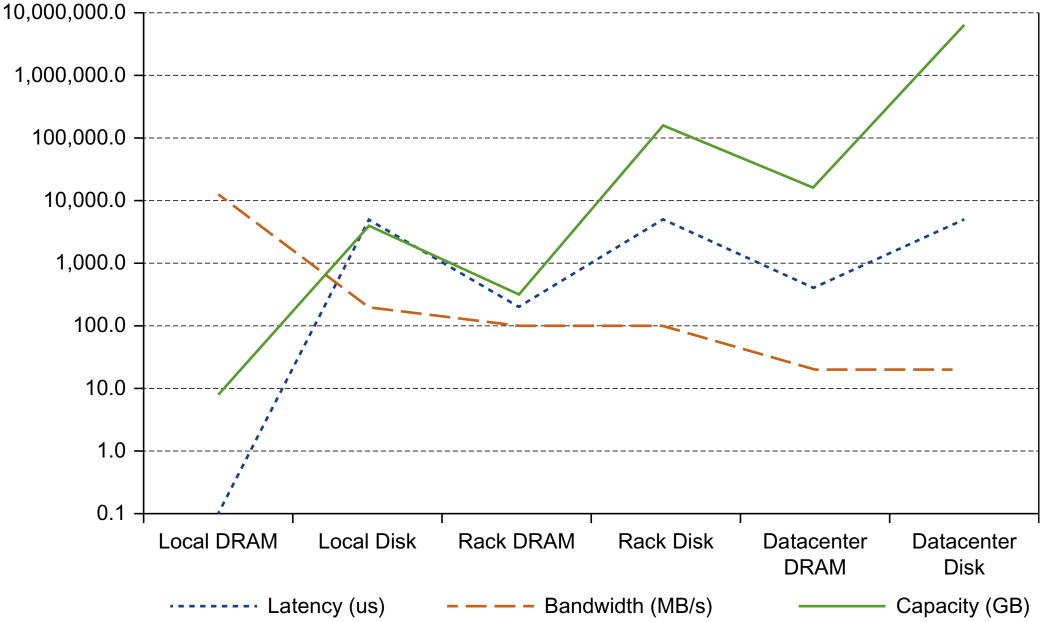


Figure 6.7 Graph of latency, bandwidth, and capacity of the memory hierarchy of a WSC for data in Figure 6.6 (Barroso et al., 2013).

- Each server contains 16 GiB of memory with a 100-ns access time and transfers at 20 GB/s, 128 GiB of Flash with 100-μs latency and transfers at 1 GB/s, and 2 TB of disk that offer a 10-ms access time and transfer at 200 MB/s. There are two sockets per board, and they share one 1 Gbit/s Ethernet port.
- In this example, every pair of racks includes one rack switch and holds 80 servers. Networking software plus switch overhead increases the latency to DRAM to 100 μs and the disk access latency to 11 ms. Thus, the total storage capacity of a rack is roughly 1 TB of DRAM, 20 TB of Flash, and 160 TB of disk storage. The 1 Gbit/s Ethernet limits the remote bandwidth to DRAM, Flash, or disk within the rack to 100 MB/s.
- The array is 30 racks, so storage capacity of an array goes up by a factor of 30: 30 TB of DRAM, 600 TB of Flash, and 4.8 PB of disk. The array switch hardware and software increases latency to DRAM within an array to 500 μs, to 600 μs for Flash, and disk latency to 12 ms. The bandwidth of the array switch limits the remote bandwidth to either array DRAM, array Flash, or array disk to 10 MB/s.

Figures 6.6 and 6.7 show that network overhead dramatically increases latency between local DRAM and Flash, rack DRAM and Flash, or array DRAM and Flash, but all still have more than 10 times better latency than accessing the local disk. The network collapses the difference in bandwidth between rack DRAM, Flash, and disk and between array DRAM, Flash, and disk.

The WSC needs 40 arrays to reach 100,000 servers, so there is one more level in the networking hierarchy. Figure 6.8 shows the conventional Layer 3 routers to connect the arrays together and to the Internet.

Most applications fit into a single array within a WSC. Those that need more than one array use *sharding* or *partitioning*, meaning that the dataset is split into independent pieces and then distributed to different arrays. As an analogy, it's like picking up registration packets for a conference with one person handling names A to M and another doing N to Z. Operations on the whole dataset are sent to the servers hosting the pieces, and the results are coalesced by the client computer.

Example What is the average memory latency assuming that 90% of accesses are local to the server, 9% are outside the server but within the rack, and 1% are outside the rack but within the array?

Answer The average memory access time is

$$(90\% \times 0.1) + (9\% \times 100) + (1\% \times 300) = 0.09 + 27 + 5 = 32.09\ \mu\text{s}$$

or a factor of more than 300 slowdown versus 100% local accesses. Clearly, locality of access within a server is vital for WSC performance.

Example How long does it take to transfer 1000 MB between disks within the server, between servers in the rack, and between servers in different racks in the array? How much faster is it to transfer 1000 MB between DRAM in the three cases?

Answer A 1000 MB transfer between disks takes

$$\text{Within server} = 1000/200 = 5\text{ s}$$

$$\text{Within rack} = 1000/100 = 10\text{ s}$$

$$\text{Within array} = 1000/10 = 100\text{ s}$$

A memory-to-memory block transfer takes

$$\text{Within server} = 1000/20000 = 0.05\text{ s}$$

$$\text{Within rack} = 1000/100 = 10\text{ s}$$

$$\text{Within array} = 1000/10 = 100\text{ s}$$

Thus, for block transfers outside a single server, it doesn't even matter whether the data are in memory or on disk because the rack switch and array switch are the bottlenecks. These performance limits affect the design of WSC software and inspire the need for higher-performance switches (see Section 6.6).

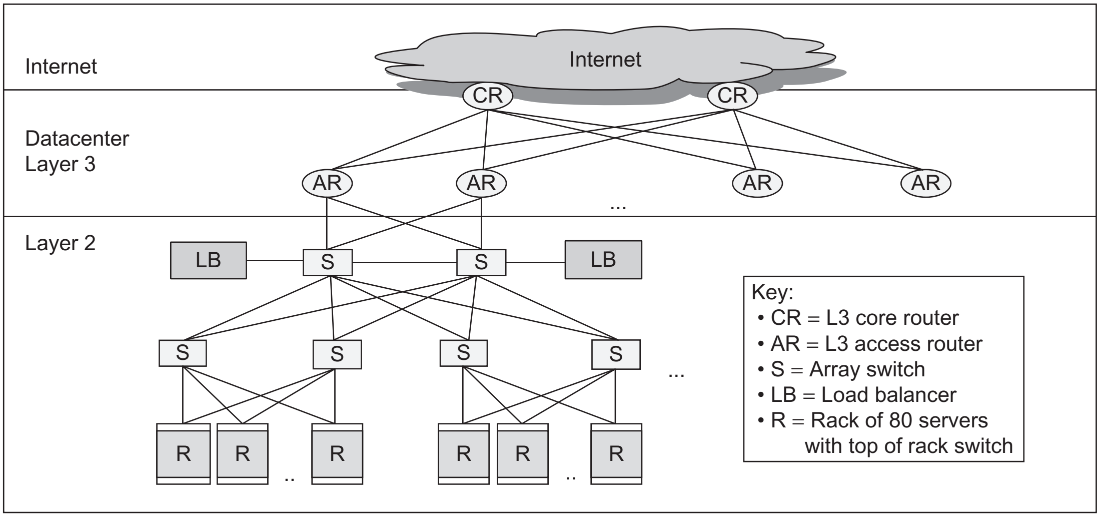

Internet
Internet
CR
CR
Datacenter Layer 3
AR
AR
AR
AR
...
Layer 2
LB
S
S
LB
S
S
S
S
...
R
R
..
R
R
R
..
R
Key:
• CR = L3 core router
• AR = L3 access router
• S = Array switch
• LB = Load balancer
• R = Rack of 80 servers with top of rack switch

Figure 6.8 A Layer 3 network used to link arrays together and to the Internet (Greenberg et al., 2009). A load balancer monitors how busy a set of servers is and directs traffic to the less loaded ones to try to keep the servers approximately equally utilized. Another option is to use a separate *border router* to connect the Internet to the data center Layer 3 switches. As we will see in Section 6.6, many modern WSCs have abandoned the conventional layered networking stack of traditional switches.

Although these examples are educational, note that computers and networking equipment can be much larger and faster than these examples from 2013 (see Section 6.7). Servers are being deployed in 2017 with 256–1024 GiB of DRAM, and recent switches have reduced delays to only 300 ns per hop.

Given the architecture of the IT equipment, we are now ready to see how to house, power, and cool it and to discuss the cost to build and operate the whole WSC, as compared to just the IT equipment within it.

6.4 The Efficiency and Cost of Warehouse-Scale Computers

Infrastructure costs for power distribution and cooling are the majority of the construction costs of a WSC, so we concentrate on them. (Section 6.7 describes the power and cooling infrastructure of a WSC in detail.)

A *computer room air-conditioning* (*CRAC*) unit cools the air in the server room using chilled water, similar to how a refrigerator removes heat by releasing it outside the refrigerator. As a liquid absorbs heat, it evaporates. Conversely, when a liquid releases heat, it condenses. Air conditioners pump the liquid into coils under low pressure to evaporate and absorb heat, which is then sent to an external condenser where it is released. Thus, in a CRAC unit, fans push warm air past a set of coils filled with cold water, and a pump moves the warmed water to the chillers to be cooled down. Figure 6.9 shows the large collection of fans and water pumps that move air and water throughout the system.

In addition to chillers, some data centers leverage colder outside air or water temperature to cool the water before it is sent to the chillers. However, depending on the location, the chillers may still be needed during the warmer times of the year.

Surprisingly, it's not obvious how to figure out how many servers a WSC can support after subtracting the overhead for power distribution and cooling. The *nameplate power rating* from the server manufacturer is always conservative: it's the maximum power a server can draw. The first step then is to measure a single server under a variety of workloads to be deployed in the WSC. (Networking is typically about 5% of power consumption, so it can be ignored at first.)

To determine the number of servers for a WSC, the available power for IT equipment could be divided just by the measured server power; however, this would again be too conservative according to Fan et al. (2007). They found that there is a significant gap between what thousands of servers could theoretically do, in the worst case, and what they will do in practice, since no real workloads will keep thousands of servers all simultaneously at their peaks. They found that they could safely oversubscribe the number of servers by as much as 40% based on the power of a single server. They recommended that WSC architects should do so to increase the average utilization of power within a WSC; however, they also suggested using extensive monitoring software along with a safety mechanism that de-schedules lower priority tasks in case the workload shifts.

Here is the power usage inside the IT equipment for a Google WSC deployed in 2012 (Barroso et al., 2013):

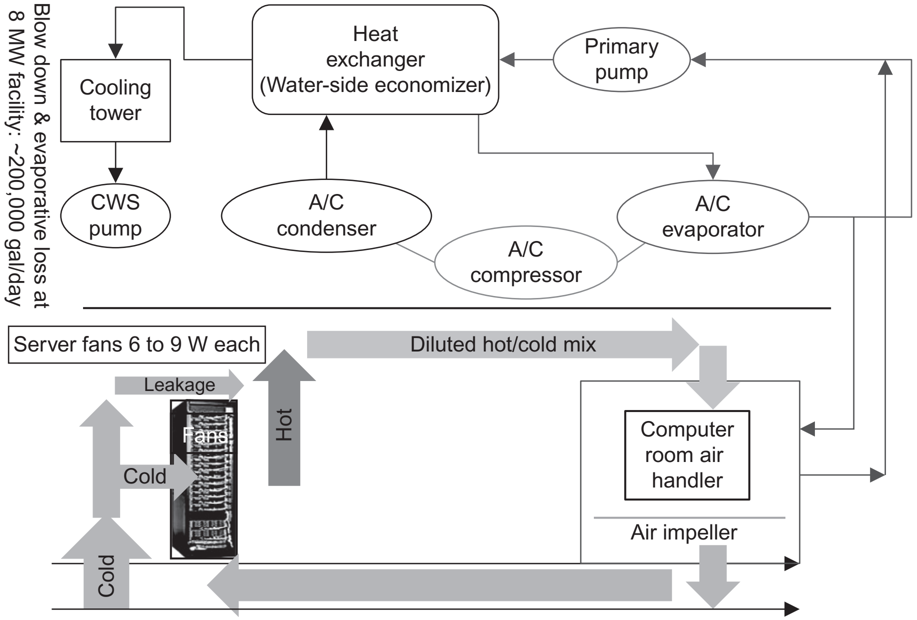

Blow down & evaporative loss at 8 MW facility: ~200,000 gal/day
Cooling tower
CWS pump
Heat exchanger (Water-side economizer)
Primary pump
A/C condenser
A/C compressor
A/C evaporator
Server fans 6 to 9 W each
Diluted hot/cold mix
Leakage
Fans
Hot
Cold
Cold
Computer room air handler
Air impeller

Figure 6.9 Mechanical design for cooling systems. CWS stands for circulating water system. From Hamilton, J., 2010. Cloud computing economies of scale. In: Paper Presented at the AWS Workshop on Genomics and Cloud Computing, June 8, 2010, Seattle, WA. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf.

- 42% of power for processors
- 12% for DRAM
- 14% for disks
- 5% for networking
- 15% for cooling overhead
- 8% for power overhead
- 4% miscellaneous

Measuring Efficiency of a WSC

A widely used, simple metric to evaluate the efficiency of a data center or a WSC is called *power utilization effectiveness* (or *PUE*):

$$\text{PUE} = (\text{Total facility power})/(\text{IT equipment power})$$

Thus, PUE must be greater than or equal to 1, and the bigger the PUE, the less efficient the WSC.

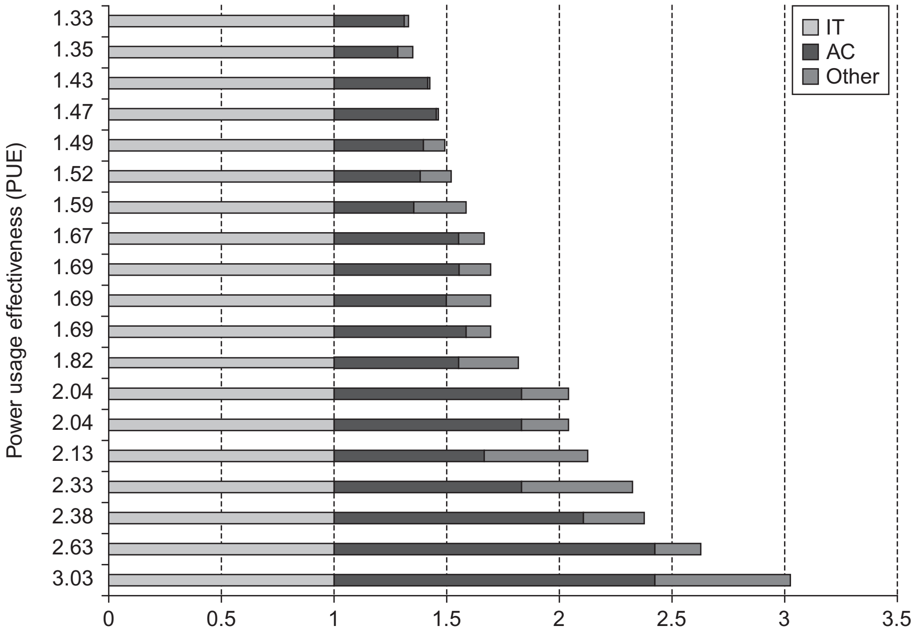


Figure 6.10 Power utilization efficiency of 19 data centers in 2006 (Greenberg et al., 2009). The power for air conditioning (AC) and other uses (such as power distribution) is normalized to the power for the IT equipment in calculating the PUE. Thus, power for IT equipment must be 1.0, and AC varies from about 0.30 to 1.40 times the power of the IT equipment. Power for "other" varies from about 0.05 to 0.60 of the IT equipment.

Greenberg et al. (2009) reported on the PUE of 19 data centers and the portion of the overhead that went into the cooling infrastructure. Figure 6.10 shows what they found, sorted by PUE from most to least efficient. The median PUE is 1.69, with the cooling infrastructure using more than half as much power as the servers—on average, 0.55 of the 1.69 is for cooling. Note that these are average PUEs, which can vary daily depending on workload and even external air temperature, as we will see (Figure 6.11).

With attention paid to PUE in the past decade, data centers are much more efficient today. However, as Section 6.8 explains, there is no universally accepted definition of what is included in PUE: If the batteries to preserve operation during a power failure are in a separate building, are they included or not? Do you measure from the output of the power substation, or where power first enters the WSC? Figure 6.10 shows the improvement in the average PUE of all Google data centers over time, which Google measures inclusively.

Since performance per dollar is the ultimate metric, we still need to measure performance. As Figure 6.7 shows, bandwidth drops and latency increases depending on the distance to the data. In a WSC, the DRAM bandwidth within a server is 200 times greater than within a rack, which in turn is 10 times greater than within an array. Thus, there is another kind of locality to consider in the placement of data and programs within a WSC.

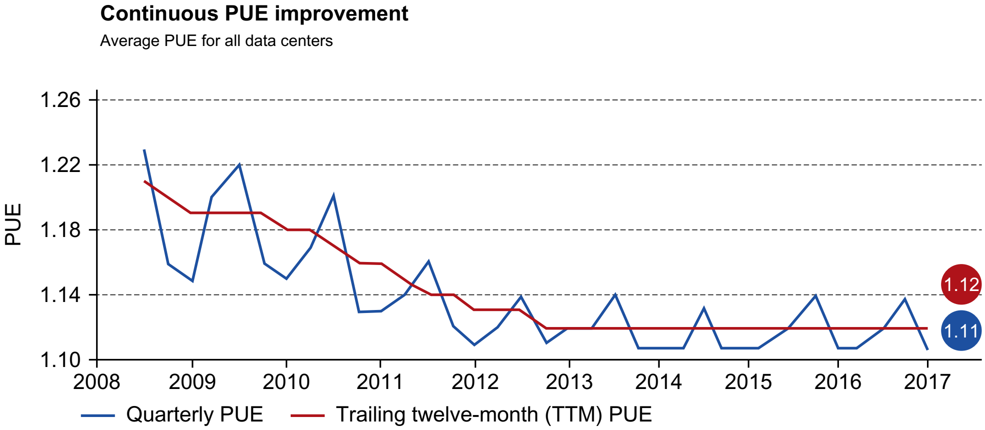


Figure 6.11 Average power utilization efficiency (PUE) of the 15 Google WSCs between 2008 and 2017. The spiking line is the quarterly average PUE, and the straighter line is the trailing 12-month average PUE. For Q4 2016, the averages were 1.11 and 1.12, respectively.

Although designers of a WSC often focus on bandwidth, programmers developing applications on a WSC are also concerned with latency because latency is visible to users. Users' satisfaction and productivity are tied to response time of a service. Several studies from the timesharing days report that user productivity is inversely proportional to time for an interaction, which was typically broken down into human entry time, system response time, and time for the person to think about the response before hitting the next entry (Doherty and Thadhani, 1982). The results of experiments showed that cutting system response time by 30% shaved the time of an interaction by 70% (Brady, 1986). This implausible result is explained by human nature: people need less time to think when given a faster response, as they are less likely to get distracted and remain "on a roll."

Figure 6.12 shows the results of a more recent experiment for the Bing search engine, where delays of 50–2000 ms were inserted at the search server (Schurman and Brutlag, 2009). As expected from previous studies, time to next click roughly doubled the delay; that is, a 200 ms delay at the server led to a 500 ms increase in time to next click. Revenue dropped linearly with increasing delay, as did user satisfaction. A separate study on the Google search engine found that these effects lingered long after the 4-week experiment ended. Five weeks later, there were 0.1% fewer searchers per day for users who experienced 200 ms delays, and there were 0.2% fewer searches by users who experienced 400 ms delays. Given the amount of money made in search, even such small changes are disconcerting. In fact, the results were so negative that they ended the experiment prematurely.

Because of this extreme concern with satisfaction of all users of an Internet service, performance goals are typically specified so that a high percentage of requests are below a latency threshold, rather than just offer a target for the average latency. Such threshold goals are called *service level objectives* (*SLOs*). An SLO might be that 99% of requests must be below 100 ms. Thus, the designers of

Server delay (ms)	Increased time to next click (ms)	Queries/ user	Any clicks/ user	User satisfaction	Revenue/ user
50	–	–	–	–	–
200	500	–	−0.3%	−0.4%	–
500	1200	–	−1.0%	−0.9%	−1.2%
1000	1900	−0.7%	−1.9%	−1.6%	−2.8%
2000	3100	−1.8%	−4.4%	−3.8%	−4.3%

Figure 6.12 Negative impact of delays at the Bing search server on user behavior (Schurman and Brutlag, 2009).

Amazon's Dynamo key-value storage system decided that for services to offer good latency on top of Dynamo, their storage system had to deliver on its latency goal 99.9% of the time (DeCandia et al., 2007). For example, one improvement of Dynamo helped the 99.9th percentile much more than the average case, which reflects their priorities.

Dean and Barroso (2013) proposed the term *tail tolerant* to describe systems designed to meet such goals:

> Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts.

The causes of unpredictability include contention for shared resources (processors networks, etc.), queuing, variable microprocessor performance because of optimizations like Turbo mode or energy-saving techniques like DVFS, software garbage collection, and many more. Google concluded that instead of trying to prevent such variability in a WSC, it made more sense to develop tail-tolerant techniques to mask or work around temporary latency spikes. For example, fine-grained load balancing can quickly move small amounts for work between servers to reduce queuing delays.

Cost of a WSC

As mentioned in the introduction, unlike most architects, designers of WSCs worry about the cost to operate as well as the cost to build the WSC. Accounting labels the former costs as *operational expenditures* (*OPEX*) and the latter costs as *capital expenditures* (*CAPEX*).

To put the cost of energy into perspective, Hamilton (2010) did a case study to estimate the costs of a WSC. He determined that the CAPEX of an 8-MW facility was $88 million and that the roughly 46,000 servers and corresponding networking equipment added another $79 million to the CAPEX for the WSC. Figure 6.13 shows the rest of the assumptions for the case study.

Size of facility (critical load watts)	8,000,000
Average power usage (%)	80%
Power usage effectiveness	1.45
Cost of power ($/kWh)	$0.07
% Power and cooling infrastructure (% of total facility cost)	82%
CAPEX for facility (not including IT equipment)	**$88,000,000**
Number of servers	45,978
Cost/server	$1450
CAPEX for servers	**$66,700,000**
Number of rack switches	1150
Cost/rack switch	$4800
Number of array switches	22
Cost/array switch	$300,000
Number of layer 3 switches	2
Cost/layer 3 switch	$500,000
Number of border routers	2
Cost/border router	$144,800
CAPEX for networking gear	**$12,810,000**
Total CAPEX for WSC	**$167,510,000**
Server amortization time	3 years
Networking amortization time	4 years
Facilities amortization time	10 years
Annual cost of money	5%

Figure 6.13 Case study for a WSC, rounded to nearest $5000. Internet bandwidth costs vary by application, so they are not included here. The remaining 18% of the CAPEX for the facility includes buying the property and the cost of construction of the building. We added people costs for security and facilities management in Figure 6.14, which were not part of the case study. Note that Hamilton's estimates were done before he joined Amazon, and they are not based on the WSC of a particular company. Based on Hamilton, J., 2010. Cloud computing economies of scale. In: Paper Presented at the AWS Workshop on Genomics and Cloud Computing, June 8, 2010, Seattle, WA. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf.

Hamilton's study works out to $11/watt for the building, power, and cooling. Barroso et al. (2013) reported consistent results for several cases, with the cost at $9 to $13/watt. Thus, a 16-MW facility costs $144 million to $208 million, *not* including the computing, storage, and networking equipment.

We can convert CAPEX into OPEX by a cost of capital conversion, assuming 5% borrowing cost, which is a standard convention in US accounting rules. That is, we can just amortize CAPEX as a fixed amount each month for the effective life of the equipment. Figure 6.14 breaks down the monthly OPEX for Hamilton's case study. Note that the amortization rates differ significantly for his case study, from 10 years for the facility to 4 years for the networking equipment and 3 years for the servers. Thus, the WSC facility lasts a decade, but the servers are replaced every 3 years and the networking equipment every 4 years. By amortizing the CAPEX, Hamilton came up with a monthly OPEX, including accounting for the cost of borrowing money (5% annually) to pay for the WSC. At \$3.8 million, the monthly OPEX is about 2% of the CAPEX (or 24% annually).

This figure allows us to calculate a handy guideline to keep in mind when making decisions about which components to use when being concerned about energy. The fully burdened cost of a watt per year in a WSC, including the cost of amortizing the power and cooling infrastructure, is

$$\frac{\text{Monthly cost of infrastructure} + \text{monthly cost of power}}{\text{Facility size in watts}} \times 12 = \frac{\$765\text{K} + \$475\text{K}}{8\text{M}} \times 12 = \$1.86$$

The cost is roughly \$2 per watt-year. Thus, reducing costs by saving energy should not result in spending more than \$2 per watt-year (see Section 6.8).

Note that in Figure 6.14, more than a third of OPEX is related to power, with that category trending up while server costs are trending down over time. The networking equipment is significant at 8% of total OPEX and 19% of the server CAPEX, and networking equipment is not trending down as quickly as servers are, perhaps because of the continuing demand for higher network bandwidth (see Figure 6.22 on page 467). This difference is especially true for the switches in the networking hierarchy above the rack, which represent most of the networking costs (see Section 6.6). People costs for security and facilities management are just 2% of OPEX. Dividing the OPEX in Figure 6.14 by the number of servers and hours per month, the cost is about \$0.11 per server per hour.

Expense (% total)	Category	Monthly cost	Percent monthly cost
Amortized CAPEX (85%)	Servers	\$2,000,000	53%
	Networking equipment	\$290,000	8%
	Power and cooling infrastructure	\$765,000	20%
	Other infrastructure	\$170,000	4%
OPEX (15%)	Monthly power use	\$475,000	13%
	Monthly people salaries and benefits	\$85,000	2%
	Total OPEX	\$3,800,000	100%

Figure 6.14 Monthly OPEX for Figure 6.13, rounded to the nearest \$5000. Note that the 3-year amortization of servers means purchasing new servers every 3 years, whereas the facility is amortized for 10 years. Thus, the amortized capital costs for servers are about three times more than for the facility. People costs include three security guard positions continuously for 24 h a day, 365 days a year, at \$20 per hour per person, and one facilities person for 24 h a day, 365 days a year, at \$30 per hour. Benefits are 30% of salaries. This calculation does not include the cost of network bandwidth to the Internet because it varies by application nor vendor maintenance fees because they vary by equipment and by negotiations.

Barroso et al. (2013) evaluated CAPEX and OPEX in terms of cost per watt per month. Thus, if a 12-MW WSC is depreciated over 12 years, the depreciation cost is \$0.08 per watt per month. They assumed the company got the capital for the WSC by taking out a loan at 8% annually—corporate loans are typically between 7% and 12%—and the interest payments added another \$0.05, giving a total of \$0.13 per watt per month. They factored in the cost of servers similarly. A 500 watt server that cost \$4000 was \$8 per watt, and the 4-year depreciation was \$0.17 per watt per month. An 8% interest on a loan for the servers added \$0.02. They estimated networking at \$0.03 per watt per month. They reported that the typical OPEX cost for multiple MW WSCs varied from \$0.02 to \$0.08 per watt per month. The grand total was \$0.37 to \$0.43 per watt per month. For an 8-MW WSC, the monthly cost minus the cost of electricity is about \$3.0 million to \$3.5 million. If we subtract the monthly power use from Hamilton's calculation, his estimate of the monthly rate will be \$3.3 million. Given the different approaches to predicting costs, these estimates are remarkably consistent.

Example The cost of electricity varies by region in the United States from \$0.03 to \$0.15 per kilowatt-hour. What is the impact on hourly server costs of these two extreme rates?

Answer We multiply the critical load of 8 MW by the average PUE from Figure 6.13 (second row) to calculate the average power usage:

$$8 \times 1.45 \times 80\% = 9.28 \text{ Megawatts}$$

The monthly cost for power then goes from \$475,000 in Figure 6.14 to \$205,000 at \$0.03 per kilowatt-hour and to \$1,015,000 at \$0.15 per kilowatt-hour. These changes in electricity cost alter the hourly server costs from \$0.11 to \$0.10 and \$0.13, respectively.

Example What would happen to monthly costs if the amortization times were all made to be the same—say, 5 years? How would that change the hourly cost per server?

Answer The spreadsheet is available online at *http://mvdirona.com/jrh/TalksAndPapers/PerspectivesDataCenterCostAndPower.xls*. Changing the amortization time to 5 years changes the first four rows of Figure 6.14 to

Servers	\$1,260,000	37%
Networking equipment	\$242,000	7%
Power and cooling infrastructure	\$1,115,000	33%
Other infrastructure	\$245,000	7%

and the total monthly OPEX is \$3,422,000. If we replaced everything every 5 years, the cost would be \$0.103 per server hour, with more of the amortized costs now being for the facility rather than the servers, as in Figure 6.14.

The rate of about \$0.10 per server per hour can be much less than the cost for many companies that own and operate their own (smaller) conventional data centers. The cost advantage of WSCs led large Internet companies to offer computing as a utility where, like electricity, you pay only for what you use. Today, utility computing is better known as *cloud computing*.

6.5 Cloud Computing: The Return of Utility Computing

> If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility.... The computer utility could become the basis of a new and important industry.
>
> **John McCarthy,**
> *MIT centennial celebration* (1961)

Driven by the demand of an increasing number of users, Internet companies such as Amazon, Google, and Microsoft built increasingly larger warehouse-scale computers from commodity components, making McCarthy's prediction eventually come true, but not as he thought because of the popularity of timesharing. This demand led to innovations in systems software to support operating at this scale, including BigTable, Colossus, Dynamo, GFS, and MapReduce. It also demanded improvement in operational techniques to deliver a service available at least 99.99% of the time despite component failures and security attacks. Examples of these techniques include failover, firewalls, virtual machines, and protection against distributed denial-of-service attacks. With the software and expertise providing the ability to scale and increasing customer demand that justified the investment, WSCs with 50,000–100,000 servers have become commonplace in 2017.

With increasing scale came increasing economies of scale. Based on a study in 2006 that compared a WSC with a data center with only 1000 servers, Hamilton (2010) reported the following advantages:

- *5.7 times reduction in storage costs*—It cost the WSC \$4.6 per GB per year for disk storage versus \$26 per GB for the data center.
- *7.1 times reduction in administrative costs*—The ratio of servers per administrator was over 1000 for the WSC versus just 140 for the data center.
- *7.3 times reduction in networking costs*—Internet bandwidth cost the WSC \$13 per Mbit/s/month versus \$95 for the data center. Not surprisingly, one can negotiate a much better price per Mbit/s by ordering 1000 Mbit/s than by ordering 10 Mbit/s.

Another economy of scale comes during purchasing. The high level of purchasing leads to volume discount prices on virtually everything in the WSC.

Economies of scale also apply to operational costs. From the prior section, we saw that many data centers operate with a PUE of 2.0. Large firms can justify hiring mechanical and power engineers to develop WSCs with lower PUEs, in the range of 1.1–1.2 (see Section 6.7).

Internet services need to be distributed to multiple WSCs both for dependability and to reduce latency, especially for international markets. All large firms use multiple WSCs for that reason. It's much more expensive for individual firms to create multiple, small data centers around the world than a single data center in their corporate headquarters.

Finally, for the reasons presented in Section 6.1, servers in data centers tend to be utilized only 10%–20% of the time. By making WSCs available to the public, uncorrelated peaks between different customers can raise average utilization above 50%.

Thus, economies of scale for a WSC offer factors of 5–7 for several components of a WSC plus a few factors of 1.5–2 for the entire WSC.

Since the last edition of this book, the concerns about security have flipped for the cloud. In 2011 there was skepticism about placing critical data in the cloud because that could make it easier for hackers to break into than if the data were kept on premises ("on prem") locked down in the local data center. In 2017 data break-ins into such data centers are so routine that they barely make the news.

For example, this insecurity has even led to rapid growth of *ransomware*—where criminals break in, encrypt all the data of an organization, and won't release the key until paid a ransom—costing firms $1 billion in 2015. In contrast, WSCs are continuously under attack, their operators respond more quickly to halt them and thus build better defenses. As a result, ransomware is unheard of inside WSCs. WSCs are clearly more secure than the vast majority of local data centers today, so many CIOs now believe that critical data is safer in the cloud than "on prem."

Although there are several cloud computing providers, we feature Amazon Web Services (AWS) since it is one of the oldest and currently the largest commercial cloud provider.

Amazon Web Services

Utility computing goes back to commercial timesharing systems and even batch processing systems of the 1960s and 1970s, where companies only paid for a terminal and a phone line and then were billed based on how much computing they used. Many efforts since the end of timesharing have tried to offer such pay-as-you-go services, but they were often met with failure.

When Amazon started offering utility computing via the Amazon Simple Storage Service (Amazon S3) and then Amazon Elastic Computer Cloud (Amazon EC2) in 2006, it made some novel technical and business decisions:

- *Virtual machines*. Building the WSC using x86-commodity computers running the Linux operating system and the Xen virtual machine solved several problems. First, it allowed Amazon to protect users from each other. Second, it simplified software distribution within a WSC, in that customers needed to install only an image and then AWS automatically distributed it to all the instances being used. Third, the ability to kill a virtual machine reliably made it easy for Amazon and customers to control resource usage. Fourth, virtual machines could limit the rate at which they used the physical processors, disks, and the network as well as the amount of main memory, which gave AWS multiple price points: the lowest price option by packing many virtual cores on a single server, the highest price option of exclusive access to all the machine resources, as well as several intermediary points. Fifth, virtual machines hid the identity of hardware, allowing AWS to continue to sell time on older machines that might otherwise be unattractive to customers if they knew the age of the machines. Finally, virtual machines allowed AWS to introduce new and faster hardware either by packing even more virtual cores per server or simply by offering instances that had higher performance per virtual core; virtualization meant that offered performance need not be an integer multiple of the performance of the hardware.

- *Very low cost*. When AWS announced a rate of $0.10 per hour per instance in 2006, it was a startlingly low amount. An instance is one virtual machine, and at $0.10 per hour, AWS allocated two instances per core on a multicore server. Thus, one EC2 computer unit is equivalent to a 1.0–1.2 GHz AMD Opteron or Intel Xeon of that era.
- *(Initial) reliance on open source software*. The availability of good-quality software that had no licensing problems or costs associated with running on hundreds or thousands of servers made utility computing much more economical for both Amazon and its customers. AWS later started offering instances including commercial third-party software at higher prices.
- *No (initial) guarantee of service*. Amazon originally promised only best effort. The low cost was so attractive that many could live without a service guarantee. Today AWS provides availability SLOs of up to 99.95% on services such as Amazon EC2 and Amazon S3. Additionally, Amazon S3 was designed for durability by saving multiple replicas of each object across multiple locations. (According to AWS, the chances of permanently losing an object are one in 100 billion.) AWS also provides a Service Health Dashboard that shows the current operational status of each of the AWS services in real time so that AWS uptime and performance are fully transparent.
- *No contract required*. In part because the costs are so low, all that is necessary to start using EC2 is a credit card.

Figures 6.15 and 6.16 show the hourly price of the many types of EC2 instances in 2017. Expanding from the 10 instance types in 2006, there are now more than 50. The fastest instance is 100 times quicker than the slowest, and

	Instance	Per hour	Ratio to m4.large	Virtual cores	Compute units	Memory (GiB)	Storage (GB)
General-purpose	t2.nano	$0.006	0.05	1	Variable	0.5	EBS only
	t2.micro	$0.012	0.11	1	Variable	1.0	EBS only
	t2.small	$0.023	0.21	1	Variable	2.0	EBS only
	t2.medium	$0.047	0.4	2	Variable	4.0	EBS only
	t2.large	$0.094	0.9	2	Variable	8.0	EBS only
	t2.xlarge	$0.188	1.7	4	Variable	16.0	EBS only
	t2.2xlarge	$0.376	3.5	8	Variable	32.0	EBS only
	m4.large	$0.108	1.0	2	6.5	8.0	EBS only
	m4.xlarge	$0.215	2.0	4	13	16.0	EBS only
	m4.2xlarge	$0.431	4.0	8	26	32.0	EBS only
	m4.4xlarge	$0.862	8.0	16	54	64.0	EBS only
	m4.10xlarge	$2.155	20.0	40	125	160.0	EBS only
	m4.16xlarge	$3.447	31.9	64	188	256.0	EBS only
	m3.medium	$0.067	0.6	1	3	3.8	1 × 4 SSD
	m3.large	$0.133	1.2	2	6.5	7.5	1 × 32 SSD
	m3.xlarge	$0.266	2.5	4	13	15.0	2 × 40 SSD
	m3.2xlarge	$0.532	4.9	8	26	30.0	2 × 80 SSD
Compute-optimized	c4.large	$0.100	0.9	2	8	3.8	EBS only
	c4.xlarge	$0.199	1.8	4	16	7.5	EBS only
	c4.2xlarge	$0.398	3.7	8	31	15.0	EBS only
	c4.4xlarge	$0.796	7.4	16	62	30.0	EBS only
	c4.8xlarge	$1.591	14.7	36	132	60.0	EBS only
	c3.large	$0.105	1.0	2	7	3.8	2 × 16 SSD
	c3.xlarge	$0.210	1.9	4	14	7.5	2 × 40 SSD
	c3.2xlarge	$0.420	3.9	8	28	15.0	2 × 80 SSD
	c3.4xlarge	$0.840	7.8	16	55	30.0	2 × 160 SSD
	c3.8xlarge	$1.680	15.6	32	108	60.0	2 × 320 SSD

Figure 6.15 Price and characteristics of on-demand general-purpose and compute-optimized EC2 instances in the Virginia region of the United States in February 2017. When AWS started, one EC2 computer unit was equivalent to a 1.0–1.2 GHz AMD Opteron or Intel Xeon of 2006. Variable instances are the newest and cheapest category. They offer the full performance of a high-frequency Intel CPU core if your workload utilizes less than 5% of the core on average over 24 h, such as for serving web pages. AWS also offers Spot Instances at a much lower cost (about 25%). With Spot Instances, customers set the price they are willing to pay and the number of instances they are willing to run, and then AWS runs the bids when the spot price drops below their level. AWS also offers Reserved Instances for cases where customers know they will use most of the instance for a year. They pay a yearly fee per instance and then an hourly rate that is about 30% of column 1 to use the service. If a Reserved Instance is used 100% for a whole year, the average cost per hour including amortization of the annual fee will be about 65% of the rate in the first column. EBS is Elastic Block Storage, which is a raw block-level storage system found elsewhere on the network, rather than in a local disk or local solid stage disk (SSD) within the same server as the VM.

	Instance	Per hour	Ratio to m4.large	Virtual cores	Compute units	Memory (GiB)	Storage (GB)
GPU	p2.xlarge	$0.900	8.3	4	12	61.0	EBS only
	p2.8xlarge	$7.200	66.7	32	94	488.0	EBS only
	p2.16xlarge	$14.400	133.3	64	188	732.0	EBS only
	g2.2xlarge	$0.650	6.0	8	26	15.0	60 SSD
	g2.8xlarge	$2.600	24.1	32	104	60.0	2 × 120 SSD
FPGA	f1.2xlarge	$1.650	15.3	8 (1 FPGA)	26	122.0	1 × 470 SSD
	f1.16xlarge	$13.200	122.2	64 (8 FPGA)	188	976.0	4 × 940 SSD
Memory-optimized	x1.16xlarge	$6.669	61.8	64	175	976.0	1 × 1920 SSD
	x1.32xlarge	$13.338	123.5	128	349	1,952.0	2 × 1920 SSD
	r3.large	$0.166	1.5	2	6.5	15.0	1 × 32 SSD
	r3.xlarge	$0.333	3.1	4	13	30.5	1 × 80 SSD
	r3.2xlarge	$0.665	6.2	8	26	61.0	1 × 160 SSD
	r3.4xlarge	$1.330	12.3	16	52	122.0	1 × 320 SSD
	r3.8xlarge	$2.660	24.6	32	104	244.0	2 × 320 SSD
	r4.large	$0.133	1.2	2	7	15.3	EBS only
	r4.xlarge	$0.266	2.5	4	14	30.5	EBS only
	r4.2xlarge	$0.532	4.9	8	27	61.0	EBS only
	r4.4xlarge	$1.064	9.9	16	53	122.0	EBS only
	r4.8xlarge	$2.128	19.7	32	99	244.0	EBS only
	r4.16xlarge	$4.256	39.4	64	195	488.0	EBS only
Storage-optimized	i2.xlarge	$0.853	7.9	4	14	30.5	1 × 800 SSD
	i2.2xlarge	$1.705	15.8	8	27	61.0	2 × 800 SSD
	i2.4xlarge	$3.410	31.6	16	53	122.0	4 × 800 SSD
	i2.8xlarge	$6.820	63.1	32	104	244.0	8 × 800 SSD
	d2.xlarge	$0.690	6.4	4	14	30.5	3 × 2000 HDD
	d2.2xlarge	$1.380	12.8	8	28	61.0	6 × 2000 HDD
	d2.4xlarge	$2.760	25.6	16	56	122.0	12 × 2000 HDD
	d2.8xlarge	$5.520	51.1	36	116	244.0	24 × 2000 HDD

Figure 6.16 Price and characteristics of on-demand GPUs, FPGAs, memory-optimized, and storage-optimized EC2 instances in the Virginia region of the United States in February 2017.

the largest offers 2000 times more memory than the smallest. Rent for the cheapest instance for a whole year is just $50.

In addition to computation, EC2 charges for long-term storage and for Internet traffic. (There is no cost for network traffic inside AWS regions.) Elastic Block Storage (EBS) costs $0.10 per GB per month when using SSDs and $0.045 per GB monthly for hard disk drives. Internet traffic costs $0.01 per GB going to EC2 and $0.09 per GB coming from EC2.

Example Calculate the cost of running the average MapReduce job in Figure 6.2 on page 438 on EC2 for several months over the years. Assume there are plenty of jobs, so there is no significant extra cost to round up so as to get an integer number of hours. Next calculate the cost per month to run all the MapReduce jobs.

Answer The first question is, what is the right size instance to match the typical server at Google? Let's assume the closest match in Figure 6.15 is a c4.large with 2 virtual cores and 3.6 GiB of memory, which costs $0.100 per hour. Figure 6.17 calculates the average and total cost per year of running the Google MapReduce workload on EC2. The average September 2016 MapReduce job would cost a little over $1 on EC2, and the total workload for that month would cost $114 million on AWS.

	Aug-04	Sep-09	Sep-12	Sep-16
Average completion time (h)	0.15	0.14	0.13	0.11
Average number of servers per job	157	156	142	130
Cost per hour of EC2 c4.large instance	$0.100	$0.100	$0.100	$0.100
Average EC2 cost per MapReduce job	$2.76	$2.23	$1.89	$1.20
Monthly number of MapReduce jobs	29,000	4,114,919	15,662,118	95,775,891
Total cost of MapReduce jobs on EC2/EBS	$80,183	$9,183,128	$29,653,610	$114,478,794

Figure 6.17 Estimated cost to run the Google MapReduce workload for select months between 2004 and 2016 (Figure 6.2) using 2017 prices for AWS EC2. Because we are using 2017 prices, these are underestimates of actual AWS costs.

Example Given the cost of MapReduce jobs, imagine that your boss wants you to investigate ways to lower costs. How much might you save using AWS Spot Instances?

Answer The MapReduce jobs could be disrupted by being kicked off a spot instance, but MapReduce is designed to tolerate and restart failed jobs. The AWS Spot price for c4.large was $0.0242 versus $0.100, which meant a savings of $87 million for September 2016, but there were no guarantees on the response times!

In addition to the low-cost and a pay-for-use model of utility computing, another strong attractor for cloud computing users is that the cloud-computing providers take on the risks of over-provisioning or under-provisioning. Because either mistake could be fatal, risk avoidance is a godsend for startup companies. If too much of the precious investment is spent on servers before the product is ready for heavy use, a company could run out of money. If the service suddenly became popular but there weren't enough servers to match the demand, a company could make a very bad impression with the potential new customers it desperately needs in order to grow.

The poster child for this scenario is FarmVille from Zynga, a social networking game on Facebook. Before FarmVille was announced, the largest social game was about five million daily players. FarmVille had one million players 4 days after launching and 10 million players after 60 days. After 270 days, it had 28 million daily players and 75 million monthly players. Because FarmVille is deployed on AWS, it is able to grow seamlessly with the number of users. Moreover, it is able to shed load based on customer demand and time of day.

FarmVille was so successful that Zynga decided to open its own data centers in 2012. In 2015, Zynga returned to AWS, deciding it was better to let AWS run its data centers (Hamilton, 2015). When FarmVille dropped from the most popular Facebook application to 110th in 2016, Zynga was able to downsize gracefully with AWS, much as it grew with AWS in the beginning.

In 2014, AWS offered a new service that hearkened back to the timesharing days of the 1960s that John McCarthy was referring to in the opening quote of this section. Instead of managing virtual machines in the cloud, *Lambda* lets users supply a function in source code (such as Python) and lets AWS automatically manage the resources required by that code to scale with input size and to make it highly available. Google Cloud Compute Functions and Microsoft Azure Functions are equivalent capabilities from competing cloud providers. As Section 6.10 explains, Google App Engine originally offered a quite similar service in 2008.

This trend is referred to as *Serverless Computing*, in that users don't have to manage servers (but these functions *are* in fact run on servers). The tasks provided include operating system maintenance, capacity provisioning and automatic scaling, code and security patch deployment, and code monitoring and logging. It runs code in response to events, such as an http request or database update. One way to think of Serverless Computing is as a set of processes running in parallel across the entire WSC that share data through a disaggregated storage service such as AWS S3.

There is no cost for Serverless Computing when a program is idle. The AWS accounting is six orders of magnitude finer than EC2, recording usage per 100 ms instead of per hour. Cost varies depending on the amount of memory needed, but if your program used 1 GiB of memory, the cost is $0.000001667 per 100 ms or about $6 per hour.

Serverless Computing can be thought of as the next evolutionary step toward realizing the cloud computing ideals of the data center as a computer, as pay-as-you-go pricing, and as a means for automatic dynamic scaling.

Cloud computing has made the benefits of WSC available to everyone. Cloud computing offers cost associativity with the illusion of infinite scalability at no extra cost to the user: 1000 servers for 1 h cost no more than 1 server for 1000 h. It is up to the cloud computing provider to ensure that there are enough servers, storage, and Internet bandwidth available to meet the demand. The previously mentioned optimized supply chain, which drops time-to-delivery to a week for new computers, is a considerable aid in providing that illusion without bankrupting the provider. This transfer of risks, cost associativity, pay-as-you-go pricing, and greater security is a powerful argument for companies of varying sizes to use cloud computing.

How Big Is the AWS Cloud?

AWS started in 2006 and grew so large that Amazon.com, rather than use a separate computing infrastructure, became one of AWS's customers in 2010. Figure 6.18 shows that AWS had facilities in 16 locations around the world in 2017, with two more on the way. As a point of interest, Figures 6.19 and 6.20 show similar maps for Google and Microsoft.

Each AWS location consists of two to three nearby facilities (one or two kilometers apart) called *availability zones*. They are so named because it should be safe to have your software running on two of them to ensure dependability as it is unlikely that both would fail simultaneously because of power outages or a natural disaster (Hamilton, 2014). These 16 locations contain 42 availability zones, and each of those zones has one or more WSCs. In 2014 each WSC had at least 50,000 servers, and some had more than 80,000.

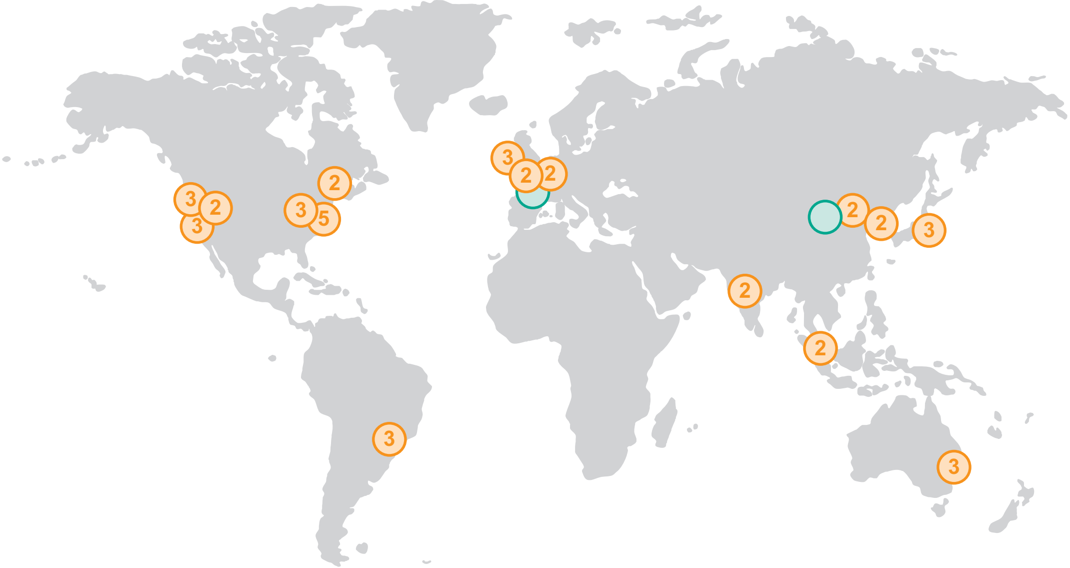


Figure 6.18 In 2017 AWS had 16 sites ("regions"), with two more opening soon. Most sites have two to three *availability zones*, which are located nearby but are unlikely to be affected by the same natural disaster or power outage, if one were to occur. (The number of availability zones are listed inside each circle on the map.) These 16 sites or regions collectively have 42 availability zones. Each availability zone has one or more WSCs. https://aws.amazon.com/about-aws/global-infrastructure/.

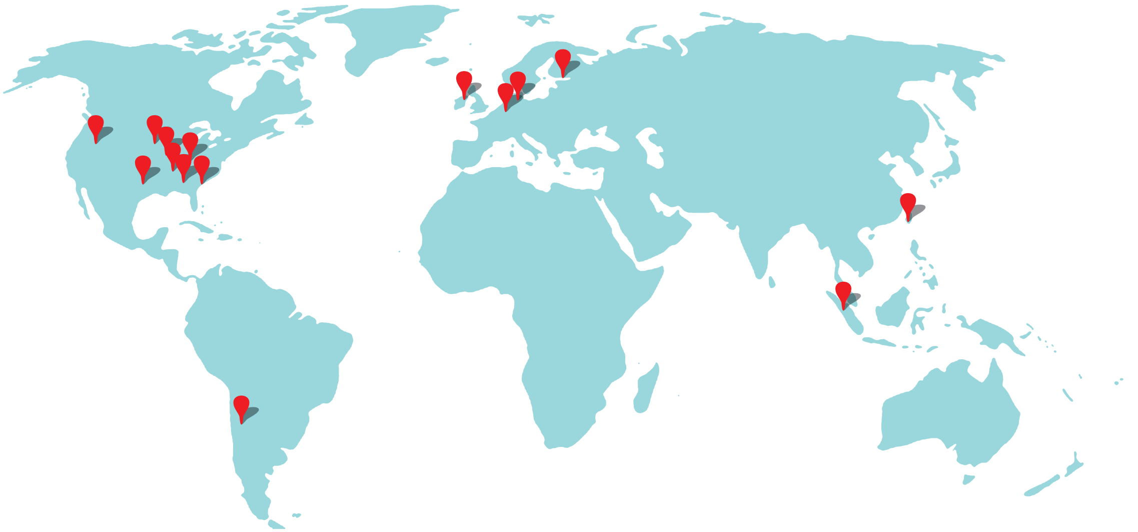


Figure 6.19 In 2017 Google had 15 sites. In the Americas: Berkeley County, South Carolina; Council Bluffs, Iowa; Douglas County, Georgia; Jackson County, Alabama; Lenoir, North Carolina; Mayes County, Oklahoma; Montgomery County, Tennessee; Quilicura, Chile; and The Dalles, Oregon. In Asia: Changhua County, Taiwan; Singapore. In Europe: Dublin, Ireland; Eemshaven, Netherlands; Hamina, Finland; St. Ghislain, Belgium. https://www.google.com/about/datacenters/inside/locations/.

Hamilton (2017) says its best to have at least three WSCs per region. The reason is simply that when one WSC fails, the other in the region needs to take on the load of the failed WSC. If there were only one other WSC, each would have to reserve half of its capacity for failover. With three, they could be used at two-thirds of capacity and still handle a quick failover. The more data centers you have, the less reserved excess capacity; AWS has regions with more than 10 WSCs.

We have found two published estimates of the total number of servers in AWS in 2014. One estimate was 2 million servers, when AWS had just 11 regions and 28 availability zones (Clark, 2014). Another estimate was between 2.8 and 5.6 million servers (Morgan 2014). If we extrapolate from 2014 to 2017 based on the increased number of availability zones, the estimates will grow to 3.0 million servers on the low end and 8.4 million on the high end. The total number of WSCs (data centers) is 84–126. Figure 6.21 shows the growth over time, using extrapolations from these two projections to offer high and low estimates of the number of severs and WSCs over time.

AWS is understandably mum on the actual number. They said that AWS had more than 1 million customers in 2014 and that "every *day* AWS adds enough physical server capacity equivalent to that needed to support Amazon.com in 2004" when it was a $7 billion annual revenue company (Hamilton, 2014).

One way to check the validity of these estimates is to look at investments. Amazon spent $24 billion in capital investments in property and equipment

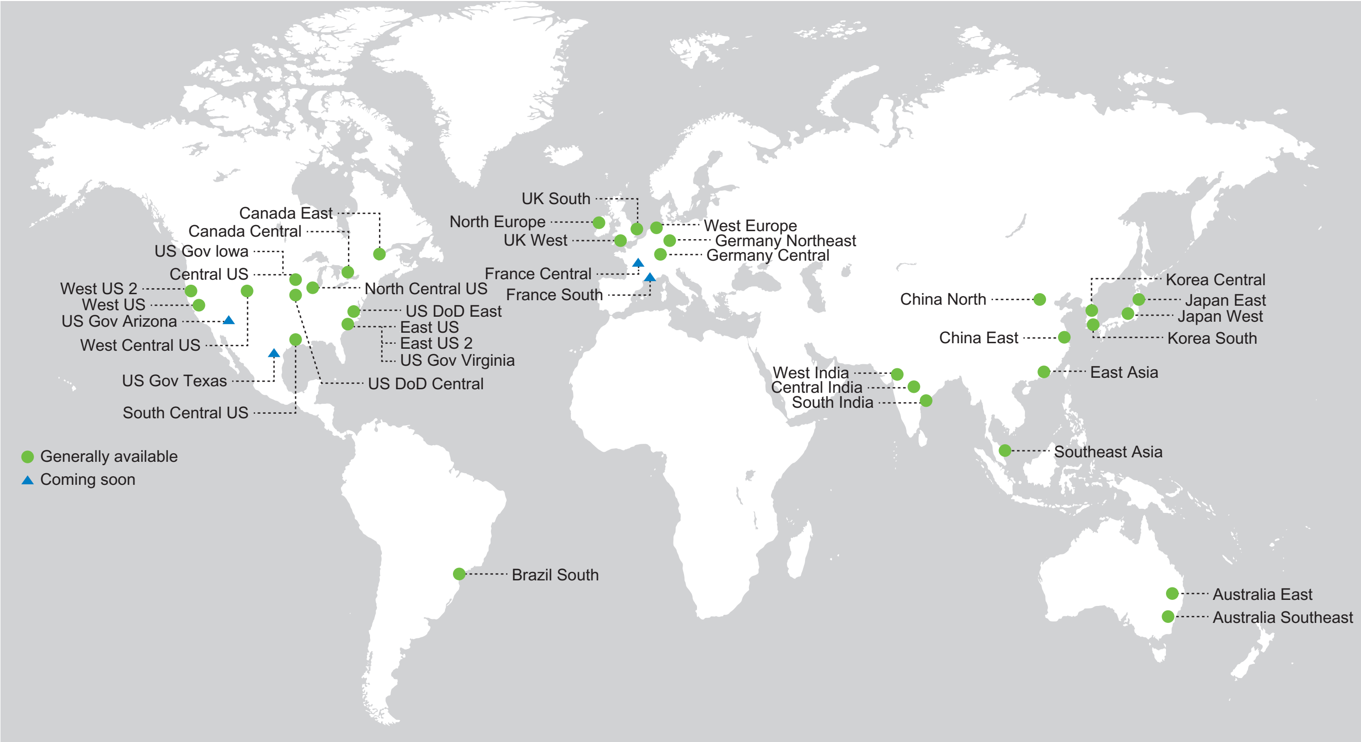

UK South
North Europe
UK West
West Europe
Germany Northeast
Germany Central
France Central
France South
Canada East
Canada Central
US Gov Iowa
Central US
West US 2
West US
US Gov Arizona
West Central US
North Central US
US DoD East
East US
East US 2
US Gov Virginia
US DoD Central
US Gov Texas
South Central US
Brazil South
China North
China East
Korea Central
Japan East
Japan West
Korea South
East Asia
West India
Central India
South India
Southeast Asia
Australia East
Australia Southeast
● Generally available
▲ Coming soon

Figure 6.20 In 2017 Microsoft had 34 sites, with four more opening soon. https://azure.microsoft.com/en-us/regions/.

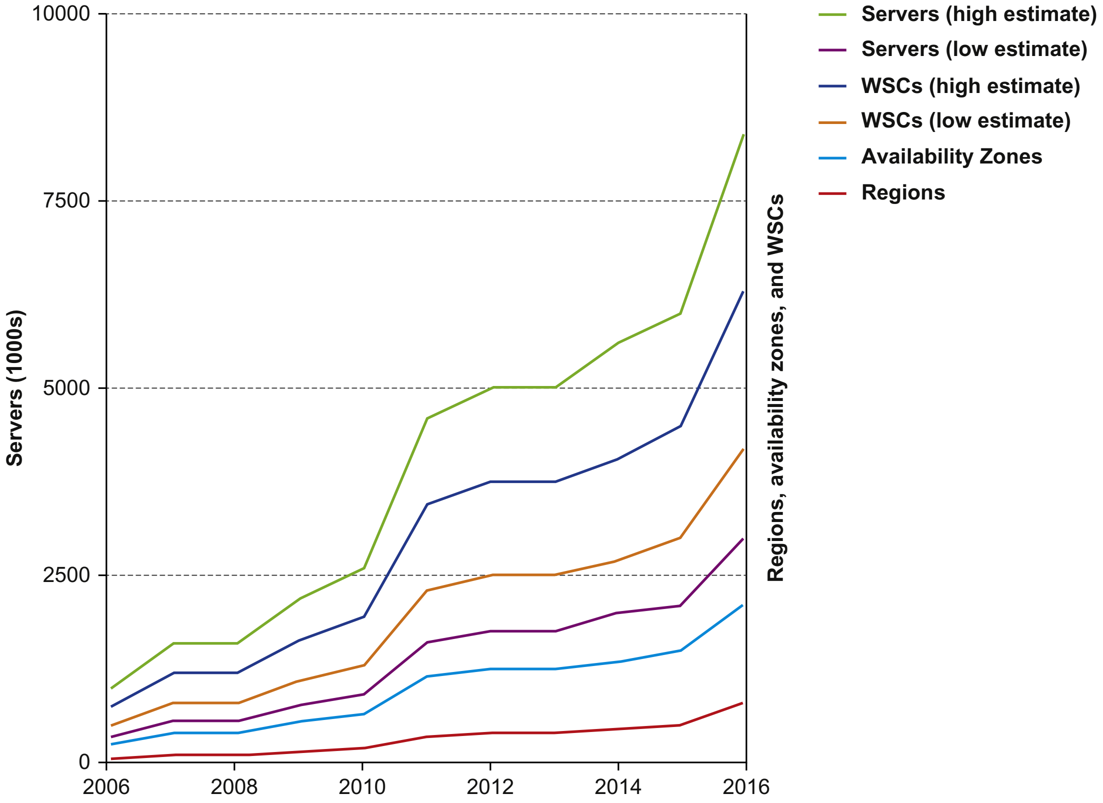


Figure 6.21 Growth of AWS regions and availability zones (right vertical axis) over time. Most regions have two or three availability zones. Each availability zone can have one or more WSCs, with the largest having more than 10 WSCs. Each WSC has at least 50,000 servers, with the biggest having more than 80,000 servers (Hamilton, 2014). Based on two published estimates for the number of AWS servers in 2014 (Clark, 2014; Morgan 2014), we project the number of servers per year (left vertical axis) and WSCs (right vertical access) as a function of the actual number of availability zones.

between 2013 and 2015, and one estimate is that two-thirds of that investment was for AWS (Gonzalez and Day 2016; Morgan 2016). Assume that it takes a year to construct a new WSC. The estimate in Figure 6.21 for 2014 to 2016 is from 34 to 51 WSCs. The cost per AWS WSC will then be \$310 million to \$470 million. Hamilton states that "even a medium sized datacenter (WSC) will likely exceed \$200M." (Hamilton, 2017). He goes on to say that today cloud providers currently have "O(10^2)" WSCs; Figure 6.21 estimate is 84–126 AWS WSCs. Despite the fuzziness of these estimates, they appear to be surprisingly consistent. He goes on to predict that to meet the future demands, the largest cloud providers will eventually rise to "O(10^5)" WSCs, or *1000* times more WSCs than today!

No matter how many servers and WSCs are in the cloud, two cross-cutting issues that shape the cost-performance of WSCs and thus cloud computing are the WSC network and the efficiency of the server hardware and software.

6.6 Cross-Cutting Issues

Net gear is the SUV of the datacenter.

James Hamilton (2009)

Preventing the WSC Network From Being a Bottleneck

Figure 6.22 shows the network demands doubling every 12–15 months for Google, resulting in a 50 × growth in traffic from the servers in Google's fleet of WSCs in just 7 years. Clearly, without great care, the WSC network could easily become a performance or cost bottleneck.

In the previous edition, we pointed out that a data center switch could cost almost $1 million, or more than 50 times as much as a Top of Rack switch. Not only was such a switch expensive, the resulting oversubscription affected the design of software and the placement of services and data within the WSC. The WSC network bottlenecks constrained data placement, which in turn complicated WSC software. Because this software is one of the most valuable assets of a WSC company, the cost of this added complexity was significant.

The ideal WSC network would be a black box whose topology and bandwidth are uninteresting because there are no restrictions: any workload could

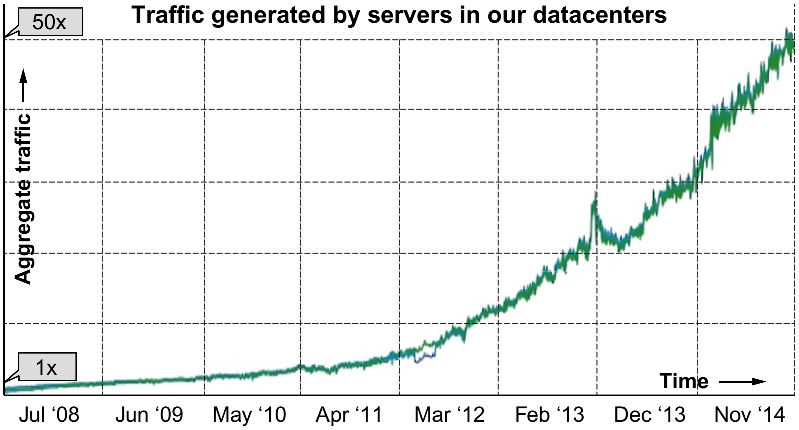


Figure 6.22 Network traffic from all the servers in Google's WSCs over 7 years (Singh et al., 2015).

be placed anywhere and optimized for server utilization rather than network traffic locality. Vahdat et al. (2010) proposed borrowing networking technology from supercomputing to overcome the price and performance problems. The latter proposed a networking infrastructure that could scale to 100,000 ports and 1 Pbit/s of bisection bandwidth. A major benefit of these novel data center switches is to simplify the software challenges because of oversubscription.

Since that time, many companies with WSC have designed their own switches to overcome these challenges (Hamilton, 2014). Singh et al. (2015) reported on several generations of custom networks used inside Google WSCs, which Figure 6.23 lists.

To keep costs down, they built their switches from standard commodity switch chips. They found that the features of traditional data center switches that were used in part to justify their high costs—such as decentralized network routing and protocols to manage support of arbitrary deployment scenarios—were unnecessary in a WSC because the network topology could be planned in advance of deployment

Data center generation switch	First deployed	Merchant silicon	Top of rack (ToR) switch config	Edge aggregation block	Spine block	Fabric speed	Host speed	Bisection BW
Four-Post CRs	2004	Vendor	48 × 1 Gbps	–	–	10 Gbps	1 Gbps	2 Tbps
Firehose 1.0	2005	8 × 10 Gbps 4 × 10 Gbps (ToR)	2 × 10 Gbps up 24 × 1 Gbps down	2 × 32 × 10 Gbps	32 × 10 Gbps	10 Gbps	1 Gbps	10 Tbps
Firehose 1.1	2006	8 × 10 Gbps	4 × 10 Gbps up 48 × 1 Gbps down	64 × 10 Gbps	32 × 10 Gbps	10 Gbps	1 Gbps	10 Tbps
Watchtower	2008	16 × 10 Gbps	4 × 10 Gbps up 48 × 1 Gbps down	4 × 128 × 10 Gbps	128 × 10 Gbps	10 Gbps	n × 1 Gbps	82 Tbps
Saturn	2009	24 × 10 Gbps	24 × 10 Gbps	4 × 288 × 10 Gbps	288 × 10 Gbps	10 Gbps	n × 10 Gbps	207 Tbps
Jupiter	2012	16 × 40 Gbps	16 × 40 Gbps	8 × 128 × 40 Gbps	128 × 40 Gbps	10/40 Gbps	n × 10 Gbps/ n × 40 Gbps	1300 Tbps

Figure 6.23 Six generations of network switches deployed at Google WSCs (Singh et al., 2015). The Four-Post CRs used commercial 512 port, 1 Gbit/s Ethernet switches, and 48-port, 1 Gbit/s Ethernet Top of Rack (ToR) switches, which allowed 20,000 servers in the array. The goal of Firehose 1.0 was to deliver 1 Gbps of nonblocking bisection bandwidth to each of 10,000 servers, but it ran into problems with the low connectivity of the ToR switch that caused problems when links failed. Firehose 1.1 was the first custom-designed switch with better connectivity in the ToR switch. Watchtower and Saturn followed in the same footsteps, but used new, faster merchant switch chips. Jupiter uses 40 Gbps links and switches to deliver more than 1 Pbit/s of bisection bandwidth. Section 6.7 describes the Jupiter switch and the Edge Aggregation and Spine Blocks of Clos networks in more detail.

and the network had only a single operator. Google instead used centralized control that relied on a common configuration that was copied to all data center switches. The modular hardware design and robust software control allowed these switches to be used both for inside the WSC and for wide area networks between WSCs. Google scaled the bandwidth of its WSCs networks by 100X in a decade, and offered more than 1 Pbit/s of bisection bandwidth in 2015.

Using Energy Efficiently Inside the Server

Although PUE measures the efficiency of a WSC, it has nothing to say about what goes on inside the IT equipment. Thus, another source of electrical inefficiency is the power supply *inside* the server, which converts input of high voltage to the voltages that chips and disks use. In 2007 many power supplies were 60%–80% efficient, which meant there were greater losses inside the server than there were going through the many steps and voltage changes from the high-voltage lines at the utility tower to supply the low-voltage lines at the server. One reason was that the power supply was often oversized in watts for what was on the motherboard. Moreover, such power supplies were typically at their worst efficiency at 25% load or less, even though, as Figure 6.3 on page 441 shows, many WSC servers operate in that range. Computer motherboards also have voltage regulator modules (VRMs), and they can have relatively low efficiency as well.

Barroso and Hölzle (2007) said the goal for the whole server should be *energy proportionality*; that is, servers should consume energy in proportion to the amount of work performed. A decade later, we've gotten close but have not hit that ideal goal. For example, the best-rated SPECpower servers in Chapter 1 still use about 20% of the full power when idle and almost 50% of full power at just 20% load. That represents huge progress since 2007 when an idle computer used 60% of full power and 70% at a 20% load, but there is still room to improve.

Systems software is designed to use all of an available resource if it potentially improves performance, without concern for the energy implications. For example, operating systems use all of memory for program data or for file caches, although much of the data will likely never be used. Software architects need to consider energy as well as performance in future designs (Carter and Rajamani, 2010).

Given the background from these six sections, we are now ready to appreciate the work of the Google WSC architects.

6.7 Putting It All Together: A Google Warehouse-Scale Computer

Because many companies with WSCs are competing vigorously in the marketplace, most have been reluctant to share their latest innovations with the public (and each other). Fortunately, Google has continued its tradition of providing details on recent WSCs for new editions of this book, once again making this edition likely the most up-to-date public description of a Google WSC, which is representative of the current state-of-the-art.

Power Distribution in a Google WSC

We start with power distribution. Although there are many variations deployed, in North America electric power typically goes through multiple voltage changes on the way to the server, starting with the high-voltage lines at the utility tower of over 110,000 V.

For large-scale sites with multiple WSCs, power is delivered to on-site substations (Figure 6.24). The substations are sized for hundreds of megawatts of power. The voltage is reduced to between 10,000 and 35,000 V for distribution to WSCs on the site.

Near the buildings of the WSC, the voltage is further reduced to around 400 V (Figure 6.25) for distribution to the rows of servers on the data center floor. (480 V is common in North America, but 400 V in the rest of the world; Google uses 415 V.) To prevent the whole WSC from going offline if power is lost, WSCs have their version of an uninterruptible power supply (UPS), just as most servers do in conventional data centers. Diesel generators are connected to the power distribution system at this level to provide power in the event of an issue with the utility power. Although most outages are less than a few minutes, WSCs store thousands of gallons of diesel on site for an extended event. The operators even make provisions with local fuel companies for continuous delivery of diesel should a site need to operate from generators for days or weeks.

Inside the WSC, power is delivered to the racks via copper bus ducts that run above each row of racks, as Figure 6.26 shows. The last step splits the three-phase power into three separate single-phase powers of 240–277 V delivered by power

Figure 6.24 An on-site substation.

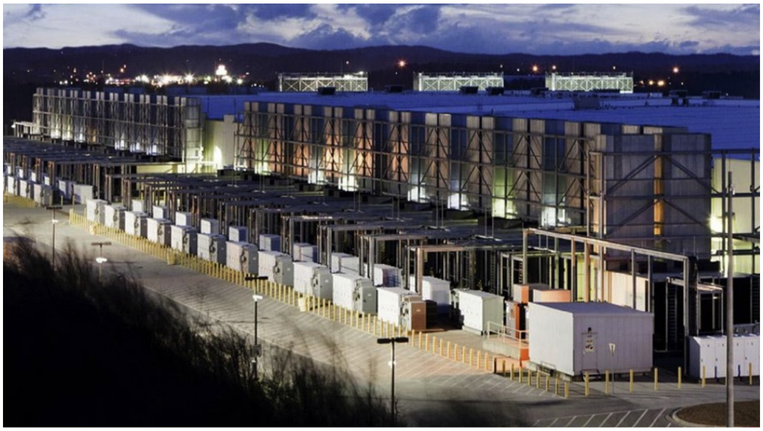


Figure 6.25 This image shows transformers, switch gear, and generators in close proximity to a WSC.

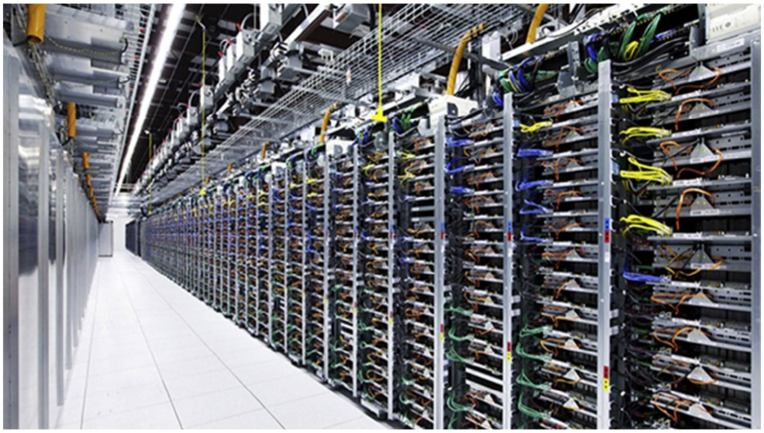


Figure 6.26 Row of servers with the copper bus ducts above that distribute 400 V to the servers. Although hard to see, they are above the shelf on the right side of the photo. It also shows a cold aisle that operators use to service the equipment.

cables to the rack. Near the top of the rack, power converters turn the 240 V AC current into 48 V DC to bring the voltage down to what boards can use.

In summary, power is distributed in a hierarchy in a WSC, with each level of the hierarchy corresponding to a distinct failure and maintenance unit: the whole WSC, arrays, rows, and racks. Software is aware of the hierarchy, and it spreads work and storage topographically to increase dependability.

WSCs around the world have different distribution voltages and frequencies, but the overall design is similar. The primary places for improvement in power efficiency are in the voltage transformers at each step, but these are highly optimized components, so there is little opportunity left.

Cooling in a Google WSC

Now that we can deliver power from the utility poles to the floor of the WSC, we need to remove the heat generated from using it. There are considerably more opportunities for improvement in the cooling infrastructure.

One of the easiest ways to improve energy efficiency is simply to run the IT equipment at higher temperatures so that the air does not need to be cooled as much. Google runs its equipment at 80+°F (27+°C), which is considerably higher than traditional data centers that are so cold that you need to wear a jacket.

Airflow is carefully planned for the IT equipment, even using Computational Fluid Dynamics simulation to design the facility. Efficient designs preserve the temperature of the cool air by reducing the chances of it mixing with hot air.

For example, most WSCs today have alternating aisles of hot air and cold air by orienting servers in opposite directions in alternating rows of racks so that hot exhaust blows in alternating directions. They are referred to as *hot aisles* and *cold aisle*s. Figure 6.26 shows a cold aisle that people use to service the servers, and Figure 6.27 shows the hot aisle. The hot air from the hot aisle rises through ducts into the ceiling.

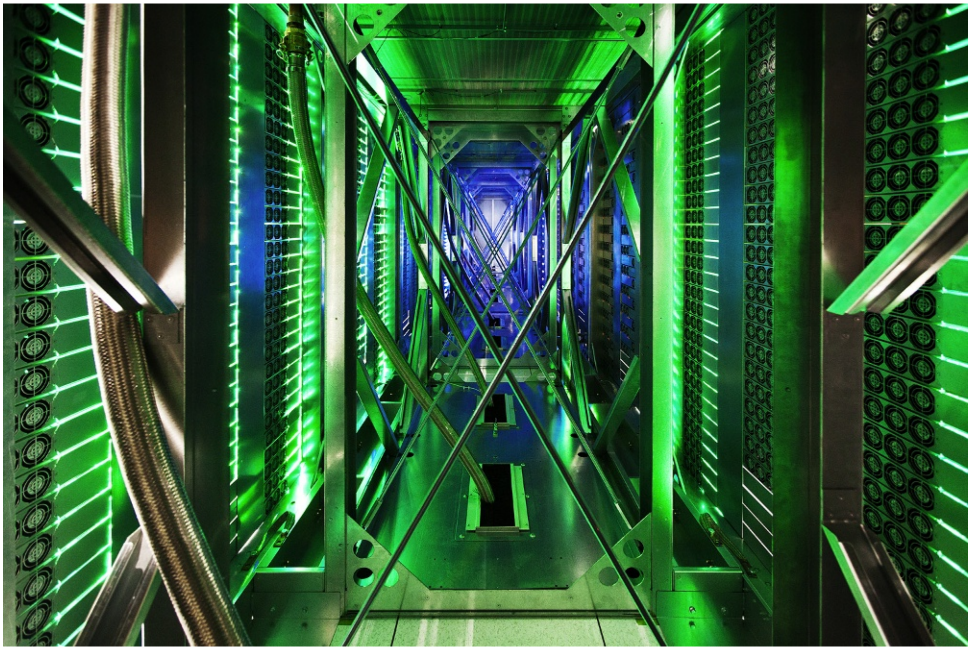


Figure 6.27 Hot aisle in a Google data center, which is clearly not designed to accommodate people.

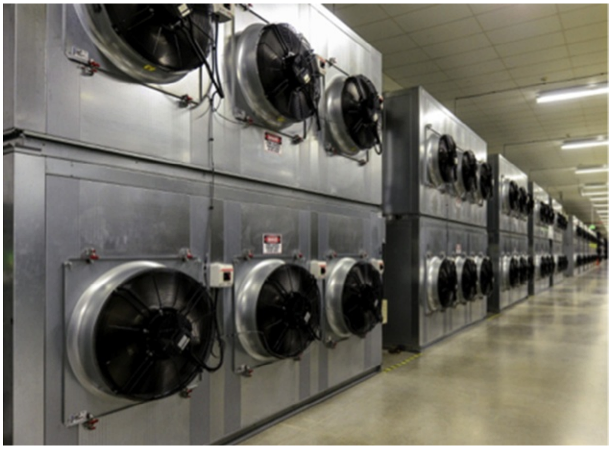


Figure 6.28 The cool air blows into the room containing the aisles of servers. The hot air goes through large vents into the ceilings where it is cooled before returning to these fans.

In conventional data centers, each server relies on internal fans to ensure a sufficient flow of cool air over the hot chips to maintain their temperature. These mechanical fans are one of the weakest components in servers; for example, the MTBF of fans is 150,000 h versus 1,200,000 h for disks. In a Google WSC, the server fans work synergistically with dozens of giant fans in the room to ensure airflow for the whole room (Figure 6.28). This division of labor means the small server fans use as little power as possible while delivering maximum performance at the worst-case power and ambient conditions. The large fans are controlled using air pressure as the control variable. The fan speeds are adjusted to maintain a minimum pressure difference between the hot and cold aisles.

To cool this hot air, they add large-scale fan-coils at either end of the rows of racks. Hot air from the racks is delivered to the fan-coils above via a horizontal plenum inside the hot aisle. (Two rows share the pair of cooling coils, as they are placed above the cold aisle between the two rows.) The cooled air is sent via a plenum in the ceiling to the wall with the big fans in Figure 6.28, which return the cooled air to the room containing the racks.

We'll describe how to remove the heat from the water in the cooling coils shortly, but let's reflect on the architecture so far. It separates the racks from the cooling capacity provided by the fan-coils, which allows for sharing of cooling across two rows of racks in the WSC. Thus, it efficiently provides more cooling to high-power racks and less to low-power racks. With thousands of racks in a WSC, they are unlikely to be identical, so power variability between racks is common, which this design accommodates.

Cool water is supplied to the individual fan-coils via a network of pipes from a cooling plant. Heat is transferred into the water via forced convection in the cooling coils, and warm water returns to a cooling plant.

To improve the efficiency of WSCs, architects try to use the local environment to remove the heat whenever possible. Evaporative *cooling towers* are common in WSCs to leverage the colder outside air to cool the water instead of it being chilled mechanically. The temperature that matters is called the *wet-bulb temperature*, which is the lowest temperature that can be achieved by evaporating water with air. It is the temperature a parcel of air would have if it were cooled to saturation (100% relative humidity) by the evaporation of water into it, with the latent heat being supplied by the parcel. Wet-bulb temperature is measured by blowing air at the bulb end of a thermometer that has water on it.

Warm water is sprayed inside in the cooling tower and collected in pools at the bottom, transferring heat to the outside air via evaporation and thereby cooling the water. This technique is called *water-side economization*. Figure 6.29 shows the steam rising above cooling towers. An alternative is to use cold water instead of crisp air. Google's WSC in Finland uses a water-to-water heat exchanger that takes the frigid water from the Gulf of Finland to chill the warm water from inside the WSC.

The cooling tower system uses water caused by evaporation in the cooling towers. For example, an 8-MW facility might need 70,000–200,000 gallons of water per day, thus the desire for the WSC to be located near ample sources of water.

Although the cooling plant is designed so that heat can be removed without artificial cooling most of the time, mechanical chillers aid in rejecting the heat in some regions when the weather is warm.

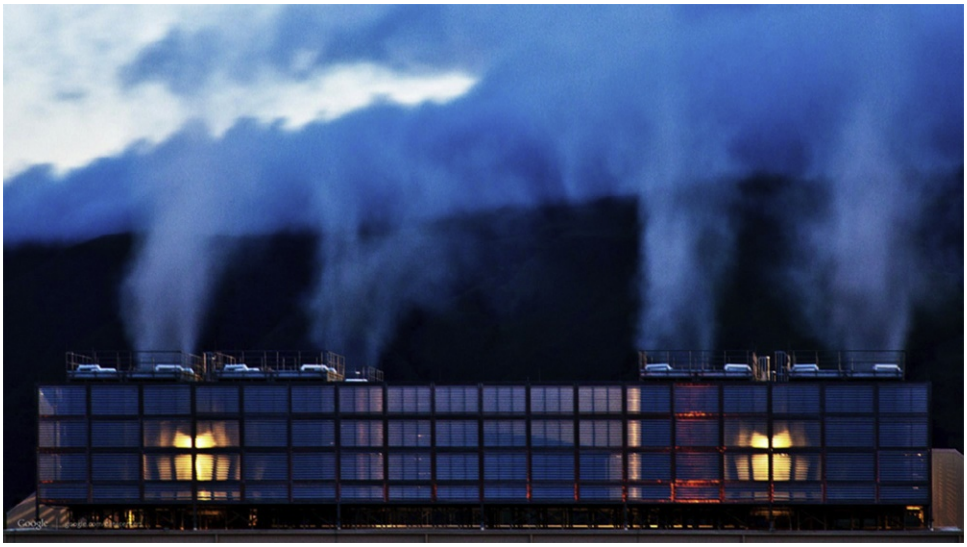


Figure 6.29 Steam rising from the cooling towers that transfer heat to the air from the water used to cool equipment.

Racks of a Google WSC

We saw how Google gets power to rack and how it cools the hot air that exhausts from the rack. Now we're ready to explore the rack itself. Figure 6.30 shows a typical rack found inside a Google WSC. To put this rack into context, a WSC consists of multiple arrays (which Google calls clusters). Although arrays vary in size, some have one to two dozen rows with each row holding two to three dozen racks.

The 20 slots shown the middle of the rack in Figure 6.30 hold the servers. Depending on their width, up to four servers can be placed in a single tray. The power converters near the top of the rack turn the 240 V AC current into 48 V DC, which is run on copper bus bars down the back of the rack to power the servers.

The diesel generators that provide backup power for the whole WSC take tens of seconds before they can offer power. Instead of populating a large room with

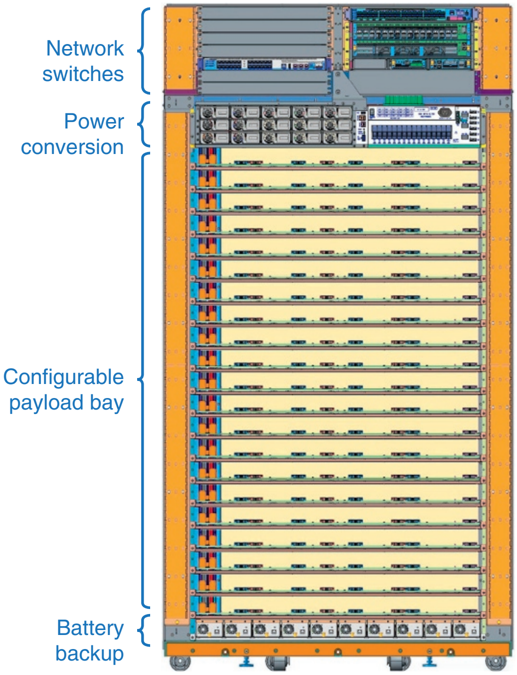


Figure 6.30 A Google rack for its WSC. Its dimensions are about 7 ft high, 4 ft wide, and 2 ft deep (2 m × 1.2 m × 0.5 m). The Top of Rack switches are indeed at the top of this rack. Next comes the power converter that converts from 240 V AC to 48 V DC for the servers in the rack using a bus bar at the back of the rack. Next is the 20 slots (depending on the height of the server) that can be configured for the various types of servers that can be placed in the rack. Up to four servers can be placed per tray. At the bottom of the rack are high-efficiency distributed modular DC uninterruptible power supply (UPS) batteries.

enough batteries to power the whole WSC for several minutes—which was a common practice in the early WSCs—Google puts small batteries at the bottom of each rack. Because UPS is distributed to each rack, the cost is incurred only as racks are deployed, instead of paying upfront for the UPS capacity of a full WSC. These batteries are also better than the traditional batteries because they are on the DC side after the voltage conversions, and they use an efficient charging scheme. In addition, replacing the 94%-efficient lead batteries with the 99.99%-efficient local UPS helps to lower the PUE. It's a very efficient UPS system.

It is comforting that the top of the rack in Figure 6.30 does indeed contain the Top of Rack switch, which we describe next.

Networking in a Google WSC

The Google WSC network uses a topology called *Clos*, which is named after the telecommunications expert who invented it (Clos, 1953). Figure 6.31 shows the structure of the Google Clos network. It is a multistage network that uses low port-count ("low radix") switches, offers fault tolerance, and increases both the network scale and its bisection bandwidth. Google increases the scale simply by adding stages to the multistage network. The fault tolerance is provided by its inherent redundancy, which means a failure of any link has only a small impact on the overall network capacity.

As Section 6.6 describes, Google builds customer switches from standard commodity switch chips and uses centralized control for network routing and management. Every switch is given a consistent copy of the current topology of the network, which simplifies the more complex routing of a Clos network.

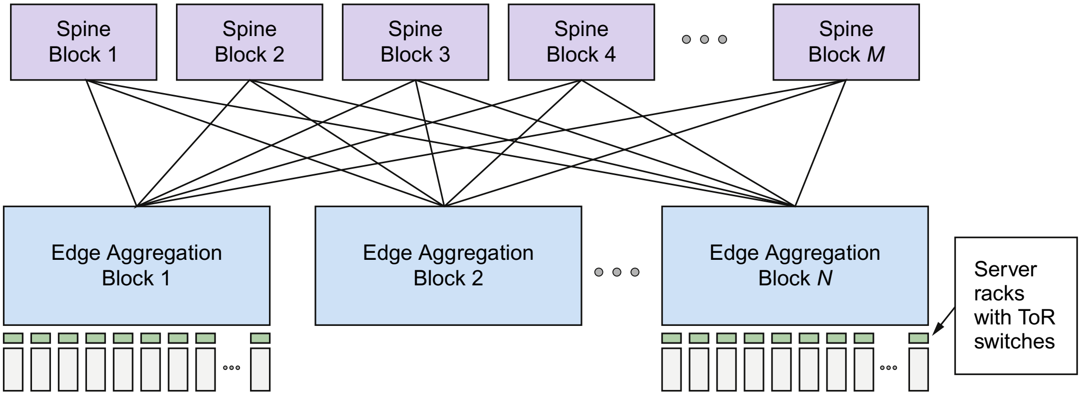


Figure 6.31 A Clos network has three logical stages containing crossbar switches: ingress, middle, and egress. Each input to the ingress stage can go through any of the middle stages to be routed to any output of the egress stage. In this figure, the middle stages are the *M* Spine Blocks, and the ingress and egress stages are in the *N* Edge Activation Blocks. Figure 6.22 shows the changes in the Spine Blocks and the Edge Aggregation Blocks over many generations of Clos networks in Google WSCs.

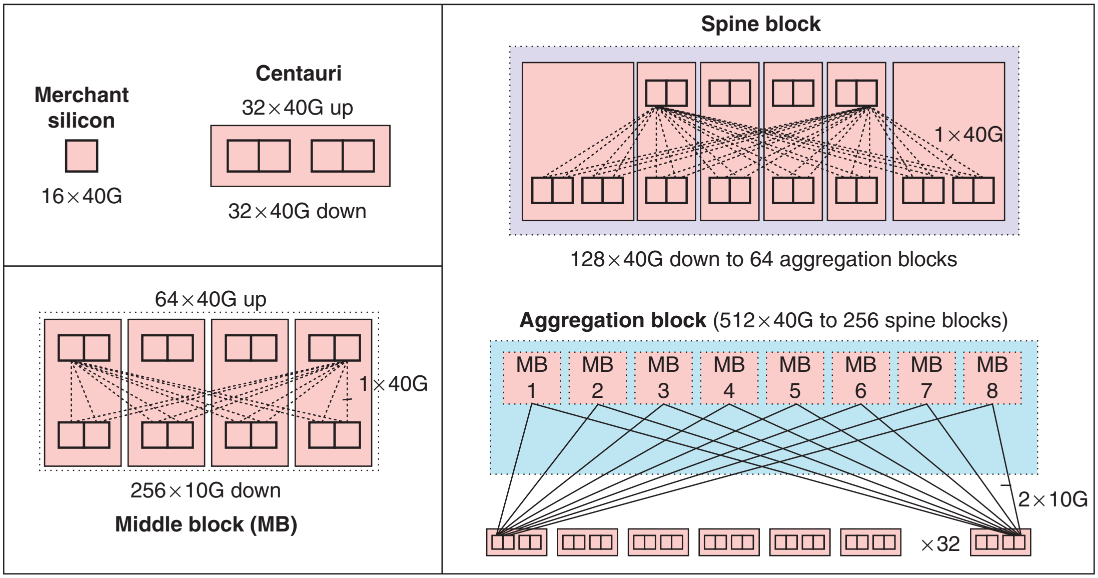

Merchant
silicon
16×40G
Centauri
32×40G up
32×40G down
64×40G up
1×40G
256×10G down
Middle block (MB)
Spine block
1×40G
128×40G down to 64 aggregation blocks
Aggregation block (512×40G to 256 spine blocks)
MB 1
MB 2
MB 3
MB 4
MB 5
MB 6
MB 7
MB 8
2×10G
×32

Figure 6.32 Building blocks of the Jupiter Clos network.

The latest Google switch is Jupiter, which is the switch's sixth generation. Figure 6.32 shows the building blocks of the switch, and Figure 6.33 shows the wiring of the middle blocks housed in racks. All the cables use bundles of optical fibers.

The commodity switch chip for Jupiter is a 16 × 16 crossbar using 40 Gbps links. The Top of Rack switch has four of these chips, which are configured with 48 40-Gbps links to the servers and 16 40-Gbps links to the network fabric, yielding an oversubscription of just 3:1, which is better than earlier generations. Moreover, this generation was the first time that servers were offered with 40-Gbps links.

The middle blocks in Figures 6.32 and 6.33 consist of 16 of the switch chips. They use two stages, with 256 10-Gbps links for the Top of Rack connectivity and 64 40-Gbps links to connect to the rest of the network fabric through the spine. Each of the chips in the Top of Rack switch connects to eight middle blocks using dual redundant 10-Gbps links.

Each aggregation block is connected to the spine block with 512 40-Gbps links. A spine block uses 24 switch chips to offer 128 40-Gbps ports to the aggregation blocks. At the largest scale, they use 64 aggregation blocks to provide dual redundant links. At this maximum size, the bisection bandwidth is an impressive 1.3 Pbit (10^{15}) per second.

Note that the whole Internet might have a bisection bandwidth of just 0.2 Pbit/s; one reason is that Jupiter was built for a high bisection bandwidth, but the Internet was not.

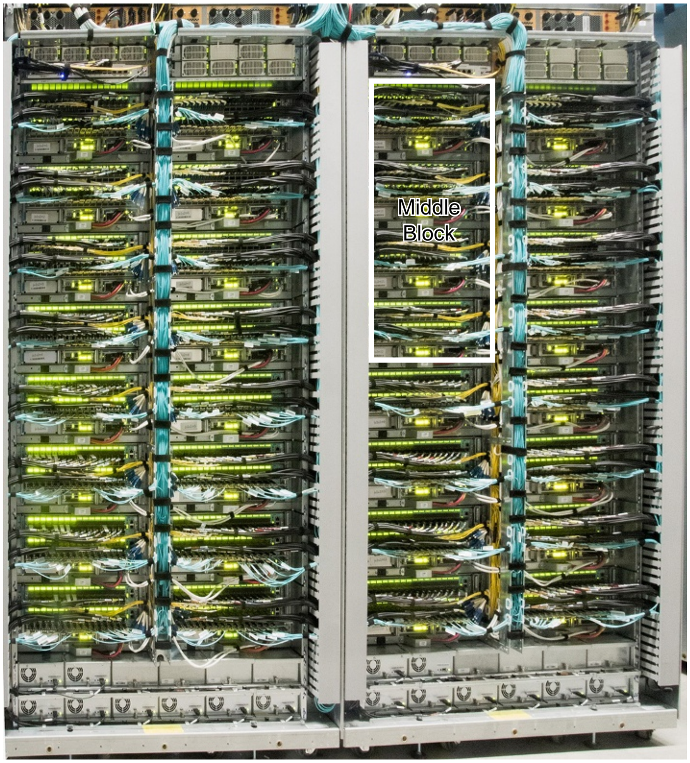


Figure 6.33 Middle blocks of the Jupiter switches housed in racks. Four are packed in a rack. A rack can hold two spine blocks.

Servers in a Google WSC

Now that we have seen how to power, cool, and communicate, we are finally ready to see the computers that do the actual work of the WSC.

The example server in Figure 6.34 has two sockets, each containing an 18-core Intel Haswell processor running at 2.3 GHz (see Section 5.8). The photo shows 16 DIMMs, and these servers are typically deployed with 256 GB total of DDR3-1600 DRAM. The Haswell memory hierarchy has two 32 KiB L1 caches, a 256 KiB L2 cache, and 2.5 MiB of L3 cache per core, resulting in a 45 MiB L3 cache. The local memory bandwidth is 44 GB/s with a latency of 70 ns, and the intersocket bandwidth is 31 GB/s with a latency of 140 ns to remote memory. Kanev et al. (2015) highlighted the differences between the SPEC benchmark suite and a WSC workload. An L3 cache is barely needed for SPEC, but it is useful for a real WSC workload.

The baseline design has a single network interface card (NIC) for a 10 Gbit/s Ethernet link, although 40 Gbit/s NICs are available. (Other cloud providers

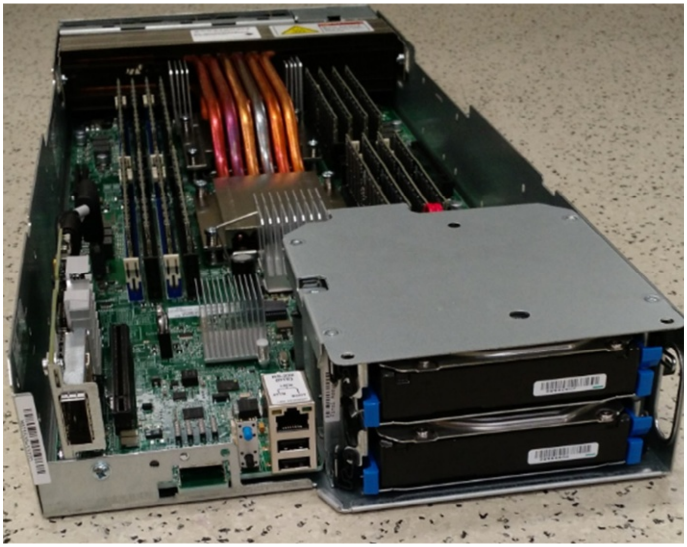


Figure 6.34 An example server from a Google WSC. The Haswell CPUs (2 sockets × 18 cores × 2 threads = 72 "virtual cores" per machine) have 2.5 MiB last level cache per core or 45 MiB using DDR3-1600. They use the Wellsburg Platform Controller Hub and have a TFP of 150 W.

moved to 25 Gbit/s or multiples thereof.) While the photo in Figure 6.34 shows two SATA disk drives, each of which can contain up to 8 TB, the server also can be configured with SSD flash drives with 1 TB of storage. The peak power of the baseline is about 150 watts. Four of these servers can fit in a slot of the rack in Figure 6.30.

This baseline node is supplemented to offer a storage (or "diskfull") node. The second unit contains 12 SATA disks and is connected to the server over PCIe. Peak power for a storage node is about 300 watts.

Conclusion

In the previous edition, the Google WSC we described had a PUE of 1.23 in 2011. As of 2017, the average PUE of the whole Google fleet of 16 sites dropped to 1.12, with the Belgium WSC leading the way with a 1.09 PUE. The energy-saving techniques include

- Operating servers at higher temperatures means that air has to be chilled only to 80+°F (27°C) instead of the traditional 64–71°F (18–22°C).
- A higher target for cold air temperature helps put the facility more often within the range that can be sustained by cooling towers, which are more energy-efficient than traditional chillers.

- Deploying WSCs in temperate climates to allow use of evaporative cooling exclusively for large portions of the year.
- Adding large fans for entire rooms to work in concert with the small fans of the servers to reduce energy while satisfying worst-case scenarios.
- Averaging the cooling per server to whole racks of servers by deploying the cooling coils per row to accommodate warmer and cooler racks.
- Deploying extensive monitoring hardware and software to measure actual PUE versus designed PUE improves operational efficiency.
- Operating more servers than the worst-case scenario for the power distribution system would suggest. It is safe since it's statistically improbable that thousands of servers would all be highly busy simultaneously as long as there is a monitoring system to off-load work in the unlikely case that they did (Fan et al., 2007; Ranganathan et al., 2006). PUE improves because the facility is operating closer to its fully designed capacity, where it is at its most efficient because the servers and cooling systems are not energy-proportional. Such increased utilization reduces demand for new servers and new WSCs.

It will be interesting to see what innovations remain to further improve the WSC efficiency so that we are good guardians of our environment. It is hard to imagine now how engineers might halve the power and cooling overhead of a WSC prior to the next edition of this book, as they did between the previous edition and this one.

6.8 Fallacies and Pitfalls

Despite WSC being just 15 years old, WSC architects like those at Google have already uncovered many pitfalls and fallacies about WSCs, often the hard way. As we said in the introduction, WSC architects are today's Seymour Crays.

Fallacy *Cloud computing providers are losing money.*

When AWS was announced, a popular question about cloud computing was whether it was profitable at the low prices at the time. Amazon Web Services has grown so large that it must be recorded separately in Amazon's quarterly reports. To the surprise of some, AWS has proved to be the most profitable portion of the company. AWS had $12.2 billion in revenue for 2016, with an operating margin of 25%, whereas Amazon's retail operations had an operating margin of less than 3%. AWS is consistently responsible for three-fourths of Amazon's profits.

Pitfall *Focusing on average performance instead of 99th percentile performance.*

As Dean and Barroso (2013) observed, developers of WSC services worry about the tail more than they care about the mean. If some customers get terrible performance, that experience can drive them away to a competitor, and they'll never return.

Pitfall *Using too wimpy a processor when trying to improve WSC cost-performance.*

Amdahl's Law still applies to WSC. There will be some serial work for each request and that can increase request latency if this work runs on a slow server (Hölzle, 2010; Lim et al., 2008). If the serial work increases latency, then the cost of using a wimpy processor must include the software development costs to optimize the code to return it to the lower latency. The larger number of threads of many slow servers can also be more difficult to schedule and load balance, and thus the variability in thread performance can lead to longer latencies. When required to wait for the longest task, a 1-in-1000 chance of bad scheduling is probably not an issue with 10 tasks, but problematic with 1000 tasks.

Many smaller servers can also lead to lower utilization because it’s clearly easier to schedule fewer things. Finally, even some parallel algorithms get less efficient when the problem is partitioned too finely. The Google rule of thumb is to use the low-end range of server class computers (Barroso and Hölzle, 2009).

As a concrete example, Reddi et al. (2010) compared embedded microprocessors (Atom) and server microprocessors (Nehalem Xeon) running the Bing search engine. They found that the latency of a query was about three times longer on Atom than on Xeon. Moreover, the Xeon was more robust. As load increases on Xeon, quality of service degrades gradually and modestly. The Atom design quickly violates its quality-of-service target as it tries to absorb additional load. Although the Atom design is more energy-efficient, the response time affects revenue, and the revenue loss is likely much greater than the cost savings of less energy. Energy-efficient designs that cannot match the response-time goals are unlikely to be deployed; we’ll see another version of this pitfall lesson in the next chapter (Section 7.9).

This behavior translates directly into search quality. Given the importance of latency to the user, as Figure 6.12 suggests, the Bing search engine uses multiple strategies to refine search results if the query latency has not yet exceeded a cutoff latency. The lower latency of the larger Xeon nodes means they can spend more time refining search results. Thus, even when the Atom had almost no load, it gave worse answers in 1% of the queries than Xeon. At normal loads, 2% of the answers were worse.

Kanev et al. (2015) has more recent, yet consistent, results.

Pitfall *Inconsistent measure of PUE by different companies.*

Google’s PUE measurements start from the power before it reaches the substation. Some measure at the entrance to the WSC, which skips voltage step downs that represent a 6% loss. There will also be different results depending on the season of the year if the WSC relies on the atmosphere to help cool the system. Finally, some report the design goal of the WSC instead of measuring the resulting system. The most conservative and best PUE measurement is a running average of the past 12 months of the measured PUE, starting from the feed of the utility.

Fallacy *Capital costs of the WSC facility are higher than for the servers that it houses.*

Although a quick look at Figure 6.13 on page 453 might lead one to that conclusion, that quick glimpse ignores the length of amortization for each part of the full WSC. However, the facility lasts 10–15 years, whereas the servers need to be repurchased every 3 or 4 years. Using the amortization times in Figure 6.13 of 10 years and 3 years, respectively, the capital expenditures over a decade are $72 million for the facility and 3.3 × $67 million, or $221 million, for servers. Thus, the capital costs for servers in a WSC over a decade are a factor of three higher than for the WSC facility.

Pitfall *Trying to save power with inactive low power modes versus active low power modes.*

Figure 6.3 on page 441 shows that the average utilization of servers is between 10% and 50%. Given the concern about operational costs of a WSC from Section 6.4, one would think low power modes would be a huge help.

As Chapter 1 mentions, DRAMs or disks cannot be accessed in these *inactive low power modes*, so they must be returned to fully active mode to read or write, no matter how low the rate. The pitfall is that the time and energy required to return to fully active mode make inactive low power modes less attractive. Figure 6.3 shows that almost all servers average at least 10% utilization, so long periods of low activity might be expected, but not long periods of inactivity (Lo et al., 2014).

In contrast, processors still run in lower power modes at a small multiple of the regular rate, so *active low power modes* are much easier to use. Note that the time to move to fully active mode for processors is also measured in microseconds, so active low power modes also address the latency concerns about low power modes.

Fallacy *Given improvements in DRAM dependability and the fault tolerance of WSC systems software, there is no need to spend extra for ECC memory in a WSC.*

Because ECC adds 8 bits to every 64 bits of DRAM, potentially a ninth of the DRAM costs could be saved by eliminating error-correcting code (ECC), especially since measurements of DRAM have claimed failure rates of 1000–5000 FIT (failures per billion hours of operation) per megabit (Tezzaron Semiconductor, 2004).

Schroeder et al. (2009) studied measurements of the DRAMs with ECC protection at the majority of Google's WSCs, which was surely many hundreds of thousands of servers, over a 2.5-year period. They found 15–25 times higher FIT rates than had been published, or 25,000–70,000 failures per megabit. Failures affected more than 8% of DIMMs, and the average DIMM had 4000 correctable errors and 0.2 uncorrectable errors per year. Measured at the server, about a third experienced DRAM errors each year, with an average of 22,000 correctable errors and 1 uncorrectable error per year. That is, for one-third of the servers, one memory error was corrected every 2.5 h. Note that these systems used the more powerful Chipkill codes rather than the simpler SECDED codes. If the easier scheme had been used, the uncorrectable error rates would have been 4–10 times higher.

In a WSC that had only parity error protection, the servers would have to reboot for each memory parity error. If the reboot time were 5 min, one-third of the machines would spend 20% of their time rebooting! Such behavior would lower the performance of the expensive facility by about 6%. Moreover, these systems would suffer many uncorrectable errors without operators being notified that they occurred.

In the early years, Google used DRAM that did not even have parity protection. In 2000, during testing before shipping the next release of the search index, it started suggesting random documents in response to test queries (Barroso and Hölzle, 2009). The reason was a stuck-at-zero fault in some DRAMs, which corrupted the new index. Google added consistency checks to detect such errors in the future. As WSC grew in size and as ECC DIMMs became more affordable, ECC became the standard in Google WSCs. ECC has the added benefit of making it much easier to find broken DIMMs during repair.

Such data suggest why the Fermi GPU (Chapter 4) adds ECC to its memory where its predecessors didn't even have parity protection. Moreover, these FIT rates cast doubts on efforts to use the Intel Atom processor in a WSC—because of its improved power efficiency—since the chip set did not support ECC DRAM.

Pitfall *Coping effectively with microsecond delays as opposed to nanosecond or millisecond delays.*

Barroso et al. (2017) point out that modern computer systems make it easy for programmers to mitigate latencies in the nanosecond and millisecond timescales (such as cache and DRAM accesses at tens of nanoseconds and disk accesses at a few milliseconds) but that such systems significantly lack support for microsecond-scale events. Programmers get a synchronous interface to the memory hierarchy, with hardware doing heroic work so that such accesses appear consistent and coherent (Chapter 2). Operating systems offer programmers a similar synchronous interface for a disk read, with many lines of OS code enabling the safe switching to another process while waiting for the disk and then returning again to the original process when the data is ready. We need new mechanisms to cope with the microsecond delays of memory technologies like Flash or the fast network interfaces like 100 Gbit/s Ethernet.

Fallacy *Turning off hardware during periods of low activity improves cost-performance of a WSC.*

Figure 6.14 on page 454 shows that the cost of amortizing the power distribution and cooling infrastructure is 50% higher than the entire monthly power bill. Thus, although it certainly would save some money to compact workloads and turn off idle machines, even if half the power were saved, the monthly operational bill would be reduced only by 7%. There would also be practical problems to overcome because the extensive WSC monitoring infrastructure depends on being able to poke equipment and see it respond. Another advantage of energy proportionality and active low power modes is that they are compatible with the WSC monitoring infrastructure, which allows a single operator to be responsible for more than 1000 servers. Note also that preventive maintenance is one of the important tasks that take place during idle time.

The conventional WSC wisdom is to run other valuable tasks during periods of little activity to recoup the investment in power distribution and cooling. A prime example is the batch MapReduce jobs that create indices for search. Another example of getting value from meager utilization is spot pricing on AWS, which the example in Figure 6.17 on page 461 illustrates. AWS users who are flexible about when their tasks are run can save up to a factor of four for computation by letting AWS schedule the tasks more flexibly using spot instances, such as when the WSC would otherwise have low utilization.

6.9 Concluding Remarks

Inheriting the title of building the world's biggest computers, computer architects of WSCs are designing the large part of the future IT that supports the mobile client and IoT devices. Many of us use WSCs many times a day, and the number of times per day and the number of people using WSCs will surely increase in the next decade. Already more than six billion of the seven billion people on the planet have cell phone subscriptions. As these devices become Internet-ready, many more people from around the world will be able to benefit from WSCs.

Moreover, the economies of scale uncovered by WSC have realized the long-dreamed-of goal of computing as a utility. Cloud computing means anyone anywhere with good ideas and business models can tap thousands of servers to deliver their vision almost instantly. Of course, there are important obstacles that could limit the growth of cloud computing around standards, privacy, the rate of growth of Internet bandwidth, and the pitfalls we mention in Section 6.8, but we foresee them being addressed so that cloud computing can continue to flourish.

Among the many attractive features of cloud computing is that it offers economic incentives for conservation. Whereas it is hard to convince cloud computing *providers* to turn off unused equipment to save energy given the cost of the infrastructure investment, it is easy to convince cloud computing *users* to give up idle instances since they are paying for them, whether or not they are doing anything useful. Similarly, charging by use encourages programmers to use computation, communication, and storage efficiently, which can be difficult to encourage without an understandable pricing scheme. The explicit pricing also makes it possible for researchers to evaluate innovations in cost-performance instead of just performance, because costs are now easily measured and believable. Finally, cloud computing means that researchers can evaluate their ideas at the scale of thousands of computers, which in the past only large companies could afford.

We believe that WSCs are changing the goals and principles of server design, just as the needs of mobile clients and IoT are changing the goals and principles of microprocessor design. Both are revolutionizing the software industry, as well. Performance per dollar and performance per joule drive both mobile client hardware and the WSC hardware, and parallelism and domain-specific accelerators are key to delivering on those sets of goals. Architects will play a vital role in both halves of this exciting future world.

Looking forward, the end of Moore's Law and Dennard scaling (Chapter 1) means that the single-thread performance of the newest processors is not that much faster than their predecessors, which will likely stretch the lifetimes of the servers in the WSCs. Thus, the money formerly spent replacing older servers will instead be used to expand to the cloud, which could mean that the cloud will be even more economically attractive in the next decade than it is today. The Moore's Law era combined with innovations in the design and operation of WSCs caused the performance-cost-energy curve of WSCs to improve continuously. With the end of that glorious era, plus the removal of the largest causes of inefficiency in WSCs, the field will likely need to look to for innovations in computer architecture of the chips that populate the WSC for sustained improvement, which is the topic of the next chapter.

6.10 Historical Perspectives and References

Section M.8 (available online) covers the development of clusters that were the foundation of WSC and of utility computing. (Readers interested in learning more should start with Barroso et al. (2013) and the blog postings of James Hamilton at *http://perspectives.mvdirona.com* plus his talks at the annual Amazon Re-Invent conference.)

Case Studies and Exercises by Parthasarathy Ranganathan

Case Study 1: Total Cost of Ownership Influencing Warehouse-Scale Computer Design Decisions

Concepts illustrated by this case study

- Total Cost of Ownership (TCO)
- Influence of Server Cost and Power on the Entire WSC
- Benefits and Drawbacks of Low-Power Servers

Total cost of ownership is an important metric for measuring the effectiveness of a warehouse-scale computer (WSC). TCO includes both the CAPEX and OPEX described in Section 6.4, and reflects the ownership cost of the entire datacenter to achieve a certain level of performance. In considering different servers, networks, and storage architectures, TCO is often the most important comparison metric used by datacenter owners to decide which options are best; however, TCO is a multidimensional computation that takes into account many different factors. The goal of this case study is to take a detailed look into WSCs, to see how different architectures influence TCO, and to understand how TCO drives operator decisions. This case study will use the numbers from Figures 6.13 and 6.14 and Section 6.4, and assumes that the described WSC achieves the operator's target level of performance. TCO is often used to compare different server options that have multiple dimensions. The exercises in this case study examine how such comparisons are made in the context of WSCs and the complexity involved in making the decisions.

6.1 [5/5/10] <6.2, 6.4> In this chapter, data-level parallelism has been discussed as a way for WSCs to achieve high performance on large problems. Conceivably, even greater performance can be obtained by using high-end servers; however, higher performance servers often come with a nonlinear price increase.

a. [5] <6.4> Assuming servers that are 10% faster at the same utilization, but are 20% more expensive, what is the CAPEX for the WSC?

b. [5] <6.4> If those servers also use 15% more power, what is the OPEX of the warehouse-scale computer?

c. [10] <6.2, 6.4> Given the speed improvement and power increase, what must the cost of the new servers be to be comparable to the original cluster? (*Hint:* Based on this TCO model, you may have to change the critical load of the facility.)

6.2 [5/10] <6.4, 6.6, 6.8> To achieve a lower OPEX, one appealing alternative is to use low-power versions of servers to reduce the total electricity required to run the servers; however, similar to high-end servers, low-power versions of high-end components also have nonlinear trade-offs.

a. [5] <6.4, 6.6, 6.8> If low-power server options offered 15% lower power at the same performance but are 20% more expensive, are they a good trade-off?

b. [10] <6.4, 6.6, 6.8> At what cost do the servers become comparable to the original cluster? What if the price of electricity doubles?

6.3 [5/10/15] <6.4, 6.6> Servers that have different operating modes offer opportunities for dynamically running different configurations in the cluster to match workload usage. Use the data in Figure 6.35 for the power/performance modes for a given low-power server.

a. [5] <6.4, 6.6> If a server operator decided to save power costs by running all servers at medium performance, how many servers would be needed to achieve the same level of performance?

b. [10] <6.4, 6.6> What are the CAPEX and OPEX of such a configuration?

Mode	Performance	Power
High	100%	100%
Medium	75%	60%
Low	59%	38%

Figure 6.35 **Power–performance modes for low-power servers.**

c. [15] <6.4, 6.6> If there was an alternative where you could purchase a server that is 20% cheaper but *x*% slower and uses *y*% less power, find the performance–power curve that provides a TCO comparable to the baseline server.

6.4 [Discussion] <6.4> Discuss the trade-offs and benefits of the two options in Exercise 6.3, assuming a constant workload being run on the servers.

6.5 [Discussion] <6.2, 6.4> Unlike high-performance computing (HPC) clusters, WSCs often experience significant workload fluctuation throughout the day. Discuss the trade-offs and benefits of the two options in Exercise 6.3, this time assuming a workload that varies.

6.6 [Discussion] <6.4, 6.7> The TCO model presented so far abstracts away a significant amount of lower level details. Discuss the impact of these abstractions to the overall accuracy of the TCO model. When are these abstractions safe to make? In what cases would greater detail provide significantly different answers?

Case Study 2: Resource Allocation in WSCs and TCO

Concepts illustrated by this case study

- Server and Power Provisioning within a WSC
- Time Variance of Workloads
- Effects of Variance on TCO

Some of the key challenges to deploying efficient WSCs are provisioning resources properly and utilizing them to their fullest capacity. This problem is complex due to the size of WSCs as well as the potential variance of the workloads being run. The exercises in this case study show how different uses of resources can affect TCO. Assume data from Figures 6.13 and 6.14 as appropriate.

6.7 [5/5/10] <6.4> One of the challenges in provisioning a WSC is determining the proper power load, given the facility size. As described in the chapter, nameplate power is often a peak value that is rarely encountered.

a. [5] <6.4> Estimate how the per-server TCO changes if the nameplate server power is 200 W and the cost is $3000.

b. [5] <6.4> Also consider a higher power, but cheaper server option whose power is 300 W and costs $2000.

c. [10] <6.4> How does the per-server TCO change if the actual average power usage of the servers is only 70% of the nameplate power?

6.8 [15/10] <6.2, 6.4> One assumption in the TCO model is that the critical load of the facility is fixed, and the amount of servers fits that critical load. In reality, due to the variations of server power based on load, the critical power used by a facility can vary at any given time. Operators must initially provision the datacenter based on its critical power resources and an estimate of how much power is used by the datacenter components.

a. [15] <6.2, 6.4> Extend the TCO model to initially provision a WSC based on a server with a nameplate power of 300 W, but also calculate the actual monthly critical power used and TCO assuming the server averages 40% utilization and so consumes only 225 W. How much capacity is left unused?

b. [10] <6.2, 6.4> Repeat this exercise with a 500-W server that averages 20% utilization and consumes only 300 W.

6.9 [10] <6.4, 6.5> WSCs are often used in an interactive manner with end users, as mentioned in Section 6.5. This interactive usage often leads to time-of-day fluctuations, with peaks correlating to specific time periods. For example, for Netflix rentals there is a peak during the evening periods of 8–10 p.m.; the entirety of these time-of-day effects is significant. Compare the per-server TCO of a datacenter with a capacity to match the utilization at 4 a.m. compared to 9 p.m.

6.10 [Discussion/15] <6.4, 6.5> Discuss some options to better utilize the excess servers during the off-peak hours or find ways to save costs. Given the interactive nature of WSCs, what are some of the challenges to aggressively reducing power usage?

6.11 [Discussion/25] <6.4, 6.6, 6.8> Propose one possible way to improve TCO by focusing on reducing server power. What are the challenges to evaluating your proposal? Estimate the TCO improvements based on your proposal. What are some advantages and drawbacks?

Exercises

6.12 [10/10/10] <6.1, 6.2> One of the important enablers of WSC is ample request-level parallelism, in contrast to instruction- or thread-level parallelism. This question explores the implication of different types of parallelism on computer architecture and system design.

a. [10] <6.1> Discuss scenarios where improving the instruction- or thread-level parallelism would provide greater benefits than those achievable through request-level parallelism.

b. [10] <6.1, 6.2> What are the software design implications of increasing request-level parallelism?

c. [10] <6.1, 6.2> What are potential drawbacks of increasing request-level parallelism?

6.13 [Discussion/15/15] <6.2, 6.3> When a cloud computing service provider receives jobs consisting of multiple Virtual Machines (VMs) (e.g., a MapReduce job), many scheduling options exist. The VMs can be scheduled in a round-robin manner to spread across all available processors and servers, or they can be consolidated to use as few processors as possible. Using these scheduling options, if a job with 24 VMs was submitted and 30 processors were available in the cloud (each able to run up to 3 VMs), round-robin would use 24 processors, while consolidated scheduling would use 8 processors. The scheduler can also find available processor cores at different scopes: socket, server, rack, and an array of racks.

a. [Discussion] <6.2, 6.3> Assuming that the submitted jobs are all compute-heavy workloads, possibly with different memory bandwidth requirements, what are the pros and cons of round-robin versus consolidated scheduling in terms of power and cooling costs, performance, and reliability?

b. [15] <6.2, 6.3> Assuming that the submitted jobs are all I/O-heavy workloads, what are the pros and cons of round-robin versus consolidated scheduling, at different scopes?

c. [15] <6.2, 6.3> Assuming that the submitted jobs are network-heavy workloads, what are the pros and cons of round-robin versus consolidated scheduling, at different scopes?

6.14 [15/15/10/10] <6.2, 6.3> MapReduce enables large amounts of parallelism by having data-independent tasks run on multiple nodes, often using commodity hardware; however, there are limits to the level of parallelism. For example, for redundancy MapReduce will write data blocks to multiple nodes, consuming disk and, potentially, network bandwidth. Assume a total dataset size of 300 GB, a network bandwidth of 1 Gb/s, a 10 s/GB map rate, and a 20 s/GB reduce rate. Also assume that 30% of the data must be read from remote nodes, and each output file is written to two other nodes for redundancy. Use Figure 6.6 for all other parameters.

a. [15] <6.2, 6.3> Assume that all nodes are in the same rack. What is the expected runtime with 5 nodes? 10 nodes? 100 nodes? 1000 nodes? Discuss the bottlenecks at each node size.

b. [15] <6.2, 6.3> Assume that there are 40 nodes per rack and that any remote read/write has an equal chance of going to any node. What is the expected runtime at 100 nodes? 1000 nodes?

c. [10] <6.2, 6.3> An important consideration is minimizing data movement as much as possible. Given the significant slowdown of going from local to rack to array accesses, software must be strongly optimized to maximize locality. Assume that there are 40 nodes per rack, and 1000 nodes are used in the MapReduce job. What is the runtime if remote accesses are within the same rack 20% of the time? 50% of the time? 80% of the time?

d. [10] <6.2, 6.3> Given the simple MapReduce program in Section 6.2, discuss some possible optimizations to maximize the locality of the workload.

6.15 [20/20/10/20/20/20] <6.2, 6.3> WSC programmers often use data replication to overcome failures in the software. Hadoop HDFS, for example, employs three-way replication (one local copy, one remote copy in the rack, and one remote copy in a separate rack), but it's worth examining when such replication is needed.

a. [20] <6.2> Let us assume that Hadoop clusters are relatively small, with 10 nodes or less, and with dataset sizes of 10 TB or less. Using the failure frequency data in Figure 6.1, what kind of availability does a 10-node Hadoop cluster have with one-, two-, and three-way replications?

b. [20] <6.2> Assuming the failure data in Figure 6.1 and a 1000-node Hadoop cluster, what kind of availability does it have with one-, two-, and three-way replications? What can you reason about the benefits of replication, at scale?

c. [10] <6.2, 6.3> The relative overhead of replication varies with the amount of data written per local compute hour. Calculate the amount of extra I/O traffic and network traffic (within and across rack) for a 1000-node Hadoop job that sorts 1 PB of data, where the intermediate results for data shuffling are written to the HDFS.

d. [20] <6.2, 6.3> Using Figure 6.6, calculate the time overhead for two- and three-way replications. Using the failure rates shown in Figure 6.1, compare the expected execution times for no replication versus two- and three-way replications.

e. [20] <6.2, 6.3> Now consider a database system applying replication on logs, assuming each transaction on average accesses the hard disk once and generates 1 KB of log data. Calculate the time overhead for two- and three-way replications. What if the transaction is executed in-memory and takes 10 μs?

f. [20] <6.2, 6.3> Now consider a database system with ACID consistency that requires two network round trips for two-phase commitment. What is the time overhead for maintaining consistency as well as replications?

6.16 [15/15/20/Discussion] <6.1, 6.2, 6.8> Although request-level parallelism allows many machines to work on a single problem in parallel, thereby achieving greater overall performance, one of the challenges is how to avoid dividing the problem too finely. If we look at this problem in the context of service level agreements (SLAs), using smaller problem sizes through greater partitioning can require increased effort to achieve the target SLA. Assume an SLA of 95% of queries respond at 0.5 s or faster, and a parallel architecture similar to MapReduce that can launch multiple redundant jobs to achieve the same result. For the following questions, assume the query–response time curve shown in Figure 6.36. The curve shows the latency of response, based on the number of queries per second, for a baseline server as well as a "small" server that uses a slower processor model.

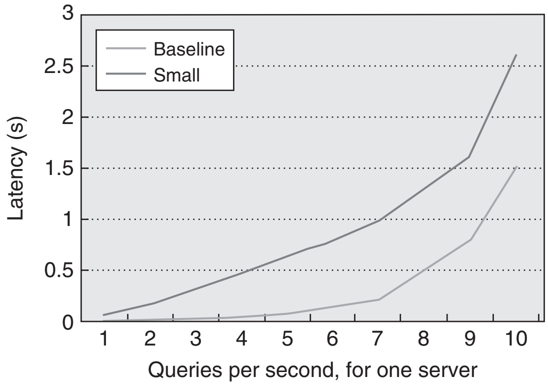


Figure 6.36 Query–response time curve.

a. [15] <6.1, 6.2, 6.8> How many servers are required to achieve this SLA, assuming the query-response time curve shown in Figure 6.36 and the WSC receiving 30,000 queries per second? How many "small" servers are required to achieve this SLA, given this response-time probability curve? Looking only at server costs, how much cheaper must the "small" servers be than the normal servers to achieve a cost advantage for the target SLA?

b. [15] <6.1, 6.2, 6.8> Often, "small" servers are also less reliable due to cheaper components. Using the numbers from Figure 6.1, assume that the number of events due to flaky machines and bad memories increases by 30%. How many "small" servers are required now? How much cheaper must those servers be than the standard servers?

c. [20] <6.1, 6.2, 6.8> Now assume a batch processing environment. The "small" servers provide 30% of the overall performance of the regular servers. Still assuming the reliability numbers from Exercise 6.15 part (b), how many "small" nodes are required to provide the same expected throughput of a 2400-node array of standard servers, assuming perfect linear scaling of performance to node size and an average task length of 10 min per node? What if the scaling is 85%? 60%?

d. [Discussion] <6.1, 6.2, 6.8> Often the scaling is not a linear function, but instead a logarithmic function. A natural response may be instead to purchase larger nodes that have more computational power per node to minimize the array size. Discuss some of the trade-offs with this architecture.

6.17 [10/10/15/Discussion] <6.3, 6.8> One trend in high-end servers is toward the inclusion of nonvolatile flash memory in the memory hierarchy, either through solid-state disks (SSDs) or PCI Express-attached cards. Typical SSDs have a bandwidth of 250 MB/s and latency of 75 μs, whereas PCIe cards have a bandwidth of 600 MB/s and latency of 35 μs.

a. [10] Take Figure 6.7 and include these points in the local server hierarchy. Assuming that identical performance scaling factors like DRAM are accessed at different hierarchy levels, how do these flash memory devices compare when accessed across the rack? Across the array?

b. [10] Discuss some software-based optimizations that can utilize the new level of the memory hierarchy.

c. [15] As discussed in "Fallacies and Pitfalls" (Section 6.8), replacing all disks with SSDs is not necessarily a cost-effective strategy. Consider a WSC operator that uses it to provide cloud services. Discuss some scenarios where using SSDs or other flash memory would make sense.

d. [Discussion] Recently, some vendors have discussed new memory technologies that are much faster than flash. As an example, look up the specifications for Intel 3D X-point memory and discuss how it would factor in Figure 6.7.

6.18 [20/20/Discussion] <6.3> *Memory Hierarchy*: Caching is heavily used in some WSC designs to reduce latency, and there are multiple caching options to satisfy varying access patterns and requirements.

a. [20] Let's consider the design options for streaming rich media from the Web (e.g., Netflix). First we need to estimate the number of videos, number of encode formats per video, and concurrent viewing users. Assume a streaming video provider that has 12,000 titles for online streaming, each title having at least four encode formats (at 500, 1000, 1600, and 2200 kbps). Let's also assume that there are 100,000 concurrent viewers for the entire site, and an average video is 75 min long (accounting for both 30-min shows and 2-h videos). Estimate the total storage capacity, I/O and network bandwidths, and video-streaming-related computation requirements.

b. [20] What are the access patterns and reference locality characteristics per user, per video, and across all videos? (*Hint*: Random versus sequential, good versus poor temporal and spatial locality, relatively small versus large working set size.)

c. [Discussion] What movie storage options exist by using DRAM, SSD, and hard drives? Compare them in performance and TCO. Would new memory technologies like those in Problem 6.17(d) be useful?

6.19 [Discussion/20/Discussion/Discussion] <6.3> Consider a social networking website with 100 million active users posting updates about themselves (in text and pictures) as well as browsing and interacting with updates in their social networks. To provide low latency, Facebook and many other websites use memcached as a caching layer before the backend storage/database tiers. Assume that at any given time the average user is browsing megabytes of content, and on any given day the average user uploads megabytes of content.

a. [20] For the social networking website discussed here, how much DRAM is needed to host its working set? Using servers each having 96 GB DRAM, estimate how many local versus remote memory accesses are needed to generate a user's home page?

b. [Discussion] Now consider two candidate memcached server designs, one using conventional Xeon processors and the other using smaller cores, such as Atom processors. Given that memcached requires large physical memory but has low CPU utilization, what are the pros and cons of these two designs?

c. [Discussion] Today's tight coupling between memory modules and processors often requires an increase in CPU socket count in order to provide large memory support. List other designs to provide large physical memory without proportionally increasing the number of sockets in a server. Compare them based on performance, power, costs, and reliability.

d. [Discussion] The same user's information can be stored in both the memcached and storage servers, and such servers can be physically hosted in different ways. Discuss the pros and cons of the following server layout in the WSC: (1)

memcached collocated on the same storage server, (2) memcached and storage servers on separate nodes in the same rack, or (3) memcached servers on the same racks and storage servers collocated on separate racks.

6.20 [5/5/10/10/Discussion/Discussion/Discussion] <6.3, 6.5, 6.6> *Datacenter Networking*: MapReduce and WSC are a powerful combination to tackle large-scale data processing. For this problem, we will assume we sort one petabyte (1 PB) of records in 6 h using 4000 servers and 48,000 hard drives (Google discussed doing this in 2008).

a. [5] Derive disk bandwidth from Figure 6.6 and associated text. How many seconds does it take to read the data into main memory and write the sorted results back?

b. [5] Assuming each server has two 1 Gb/s Ethernet network interface cards (NICs) and the WSC switch infrastructure is oversubscribed by a factor of 4, how many seconds does it take to shuffle the entire dataset across 4000 servers?

c. [10] Assuming network transfer is the performance bottleneck for petabyte sort, can you estimate what oversubscription ratio Google has in its datacenter?

d. [10] Now let's examine the benefits of having 10 Gb/s Ethernet without oversubscription—for example, using a 48-port 10 Gb/s Ethernet (this was used by the 2010 Indy sort benchmark winner TritonSort). How long does it take to shuffle 1 PB of data?

e. [Discussion] Compare the two approaches here: (1) the massive scale-out approach with high network oversubscription ratio, and (2) a relatively small-scale system with a high-bandwidth network. What are their potential bottlenecks? What are their advantages and disadvantages, in terms of scalability and TCO?

f. [Discussion] Sort and many important scientific computing workloads are communication-heavy, while many other workloads are not. List three example workloads that do not benefit from high-speed networking. What EC2 instances would you recommend to use for these two classes of workloads?

g. [Discussion] Look up the various benchmarks in www.sortbenchmark.org and recent winners in each category. How do these results match the insights from the discussion in part (e) above? How does the cloud instance used for the most recent winner of CloudSort compare with your answer in part (f) above?

6.21 [10/25/Discussion] <6.4, 6.6> Because of the massive scale of WSCs, it is very important to properly allocate network resources based on the workloads that are expected to be run. Different allocations can have significant impacts on both the performance and total cost of ownership.

a. [10] Using the numbers in the spreadsheet detailed in Figure 6.13, what is the oversubscription ratio at each access-layer switch? What is the impact on TCO if the oversubscription ratio is cut in half? What if it is doubled?

b. [25] Reducing the oversubscription ratio can potentially improve performance if a workload is network-limited. Assume a MapReduce job that uses 120 servers and reads 5 TB of data. Assume the same ratio of read/intermediate/output data as in Figure 6.2, Sep-09, and use Figure 6.6 to define the bandwidths of the memory hierarchy. When reading data, assume that 50% of data is read from remote disks; of that, 80% is read from within the rack and 20% is read from within the array. For intermediate data and output data, assume that 30% of the data uses remote disks; of that, 90% is within the rack and 10% is within the array. What is the overall performance improvement when reducing the oversubscription ratio by half? What is the performance if the oversubscription ratio is doubled? Calculate the TCO in each case.

c. [Discussion] We are seeing the trend to more cores per system. We are also seeing the increasing adoption of optical communication (with potentially higher bandwidth and improved energy efficiency). How do you think these and other emerging technology trends will affect the design of future WSCs?

6.22 [5/15/15/20/25/Discussion] <6.5> *Realizing the Capability of the Cloud*: Imagine you are the site operation and infrastructure manager of an Alexa.com top site and are considering using Amazon Web Services (AWS). What factors do you need to consider in determining whether to migrate to AWS? What services and instance types could you use, and how much cost could you save? You can use Alexa and site traffic information (e.g., Wikipedia provides page view stats) to estimate the amount of traffic received by a top site, or you can take concrete examples from the Web, such as the following example: *http://2bits.com/sites/2bits.com/files/drupal-single-server-2.8-million-page-views-a-day.pdf*. The slides describe an Alexa #3400 site that receives 2.8 million page views per day, using a single server. The server has two quad-core Xeon 2.5 GHz processors with 8 GB DRAM and three 15 K RPM SAS hard drives in a RAID1 configuration, and it costs about $400 per month. The site uses caching heavily, and the CPU utilization ranges from 50% to 250% (roughly 0.5–2.5 cores busy).

a. [5] Looking at the available EC2 instances *(http://aws.amazon.com/ec2/instance-types/)*, what instance types match or exceed the current server configuration?

b. [15] Looking at the EC2 pricing information *(http://aws.amazon.com/ec2/pricing/)*, select the most cost-efficient EC2 instances (combinations allowed) to host the site on AWS. What is the monthly cost for EC2?

c. [15] Now add the costs for IP address and network traffic to the equation, and suppose the site transfers 100 GB/day in and out on the Internet. What is the monthly cost for the site now?

d. [20] AWS also offers a micro instance for free for 1 year to new customers and 15 GB bandwidth each for traffic going in and out across AWS. Based on your estimation of peak and average traffic from your department Web server, can you host it for free on AWS?

e. [25] Based on the service characteristics, if a much larger site like Netflix.com migrates its streaming and encoding infrastructure to AWS, what AWS services could be used by Netflix and for what purposes?

f. [Discussion] Look at similar offerings from other cloud providers (Google, Microsoft, Alibaba, etc.). How do the answers to parts (a)–(e) change?

g. [Discussion] "Serverless computing" allows you to build and run higher-level applications and services without thinking about specific servers. Examples include AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, etc. Continuing to wear your site operation and infrastructure manager hat, when would you consider serverless computing?

6.23 [Discussion/Discussion/20/20/Discussion] <6.4, 6.8> Figure 6.12 shows the impact of user perceived response time on revenue, and motivates the need to achieve high-throughput while maintaining low latency.

a. [Discussion] Taking Web search as an example, what are the possible ways of reducing query latency?

b. [Discussion] What monitoring statistics can you collect to help understand where time is spent? How do you plan to implement such a monitoring tool?

c. [20] Assuming that the number of disk accesses per query follows a normal distribution, with an average of 2 and standard deviation of 3, what kind of disk access latency is needed to satisfy a latency SLA of 0.1 s for 95% of the queries?

d. [20] In-memory caching can reduce the frequencies of long-latency events (e.g., accessing hard drives). Assuming a steady-state hit rate of 40%, hit latency of 0.05 s, and miss latency of 0.2 s, does caching help meet a latency SLA of 0.1 s for 95% of the queries?

e. [Discussion] When can cached content become stale or even inconsistent? How often can this happen? How can you detect and invalidate such content?

6.24 [15/15/20/Discussion] <6.4, 6.6> The efficiency of typical power supply units (PSUs) varies as the load changes; for example, PSU efficiency can be about 80% at 40% load (e.g., output 40 W from a 100-W PSU), 75% when the load is between 20% and 40%, and 65% when the load is below 20%.

a. [15] Assume a power-proportional server whose actual power is proportional to CPU utilization, with a utilization curve as shown in Figure 6.3. What is the average PSU efficiency?

b. [15] Suppose the server employs 2 N redundancy for PSUs (i.e., doubles the number of PSUs) to ensure stable power when one PSU fails. What is the average PSU efficiency?

c. [20] Blade server vendors use a shared pool of PSUs not only to provide redundancy but also to dynamically match the number of PSUs to the server's actual power consumption. The HP c7000 enclosure uses up to six PSUs for a total of 16 servers. In this case, what is the average PSU efficiency for the enclosure of server with the same utilization curve?

d. [Discussion] Consider the impact of the different efficiency numbers in the context of the broader TCO discussions in Figures 6.13 and 6.14: how do the different design impact the total TCO? Given these, how would you optimize designs for future warehouse-scale computers?

6.25 [5/Discussion/10/15/Discussion/Discussion/Discussion] <6.4, 6.8> *Power stranding* is a term used to refer to power capacity that is provisioned but not used in a datacenter. Consider the data presented in Figure 6.37 [Fan, Weber, and Barroso, 2007] for different groups of machines. (Note that what this paper calls a “cluster” is what we have referred to as an “array” in this chapter.)

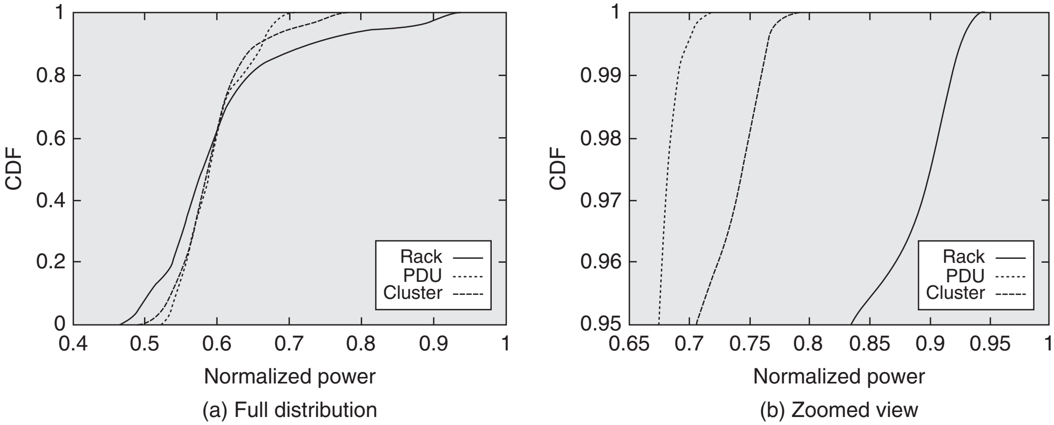
a. [5] What is the stranded power at (1) the rack level, (2) the power distribution unit level, and (3) the array (cluster) level? What are the trends with oversubscription of power capacity at larger groups of machines?

b. [Discussion] What do you think causes the differences between power stranding at different groups of machines?

c. [10] Consider an array-level collection of machines where the total machines never use more than 72% of the aggregate power (this is sometimes also referred to as the ratio between the peak-of-sum and sum-of-peaks usage). Using the cost model in the case study, compute the cost savings from comparing a datacenter provisioned for peak capacity and one provisioned for actual use.

d. [15] Assume that the datacenter designer chose to include additional servers at the array level to take advantage of the stranded power. Using the example configuration and assumptions in part (a), compute how many more servers can now be included in the warehouse-scale computer for the same total power provisioning.

e. [Discussion] What is needed to make the optimization of part (d) work in a real-world deployment? (*Hint*: Think about what needs to happen to cap power in the rare case when all the servers in the array are used at peak power.)



(a) Full distribution

(b) Zoomed view

Figure 6.37 Cumulative distribution function (CDF) of a real datacenter.

f. [Discussion] Two kinds of policies can be envisioned to manage power caps [Ranganathan et al., 2006]: (1) preemptive policies where power budgets are predetermined ("don't assume you can use more power; ask before you do!") or (2) reactive policies where power budgets are throttled in the event of a power budget violation ("use as much power as needed until told you can't!"). Discuss the trade-offs between these approaches and when you would use each type.

g. [Discussion] What happens to the total stranded power if systems become more energy-proportional (assume workloads similar to that of Figure 6.4)?

6.26 [5/20/Discussion] <6.4, 6.7> Section 6.7 discussed the use of per-server battery sources in the Google design. Let us examine the consequences of this design.

a. [5] Assume that the use of a battery as a mini-server-level UPS is 99.99% efficient and eliminates the need for a facility-wide UPS that is only 92% efficient. Assume that substation switching is 99.7% efficient and that the efficiency for the PDU, step-down stages, and other electrical breakers are 98%, 98%, and 99%, respectively. Calculate the overall power infrastructure efficiency improvements from using a per-server battery backup.

b. [20] Assume that the UPS is 10% of the cost of the IT equipment. Using the rest of the assumptions from the cost model in the case study, what is the break-even point for the costs of the battery (as a fraction of the cost of a single server) at which the total cost of ownership for a battery-based solution is better than that for a facility-wide UPS?

c. [Discussion] What are the other trade-offs between these two approaches? In particular, how do you think the manageability and failure model will change across these two different designs?

6.27 [5/5/Discussion] <6.4> For this exercise, consider a simplified equation for the total operational power of a WSC as follows:

$$\text{Total operational power} = (1 + \text{Cooling inefficiency multiplier}) * \text{IT equipment power}.$$

a. [5] Assume an 8 MW datacenter at 80% power usage, electricity costs of \$0.10 per kilowatt-hour, and a cooling-inefficiency multiplier of 0.8. Compare the cost savings from (1) an optimization that improves cooling efficiency by 20%, and (2) an optimization that improves the energy efficiency of the IT equipment by 20%.

b. [5] What is the percentage improvement in IT equipment energy efficiency needed to match the cost savings from a 20% improvement in cooling efficiency?

c. [Discussion/10] What conclusions can you draw about the relative importance of optimizations that focus on server energy efficiency and cooling energy efficiency?

6.28 [5/5/Discussion] <6.4> As discussed in this chapter, the cooling equipment in WSCs can themselves consume a lot of energy. Cooling costs can be lowered by proactively managing temperature. Temperature-aware workload placement is one optimization that has been proposed to manage temperature to reduce cooling costs. The idea is to identify the cooling profile of a given room and map the hotter systems to the cooler spots, so that at the WSC level the requirements for overall cooling are reduced.

a. [5] The coefficient of performance (COP) of a computer room air conditioning unit (CRAC) is a measure of its efficiency, and is defined as the ratio of heat removed (Q) to the amount of work necessary (W) to remove that heat. The COP of a CRAC unit increases with the temperature of the air the CRAC unit pushes into the plenum. If air returns to the CRAC unit at 20°C and we remove 10 KW of heat with a COP of 1.9, how much energy do we expend in the CRAC unit? If it takes a COP of 3.1 to cool the same volume of air, but the air is returned at 25°C, how much energy do we now expend in the CRAC unit?

b. [5] Assume a workload distribution algorithm is able to match the hot workloads well with the cool spots to allow the computer room air-conditioning (CRAC) unit to be run at a higher temperature to improve cooling efficiencies like in the exercise above. What is the power savings between the two cases described above?

c. [Discussion] Given the scale of WSC systems, power management can be a complex, multifaceted problem. Optimizations to improve energy efficiency can be implemented in hardware and in software, at the system level, and at the cluster level for the IT equipment or the cooling equipment, etc. It is important to consider these interactions when designing an overall energy-efficiency solution for the WSC. Consider a consolidation algorithm that looks at server utilization and consolidates different workload classes on the same server to increase server utilization (this can potentially have the server operating at a higher energy efficiency if the system is not energy-proportional). How would this optimization interact with a concurrent algorithm that tried to use different power states (see ACPI, Advanced Configuration Power Interface, for some examples)? What other examples can you think of where multiple optimizations can potentially conflict with one another in a WSC? How would you solve this problem?

6.29 [5/10/15/20] <6.2, 6.6> Energy proportionality (sometimes also referred to as energy scale-down) is the attribute of the system to consume no power when idle, but more importantly gradually consume more power *in proportion* to the activity level and work done. In this exercise, we will examine the sensitivity of energy consumption to different energy proportionality models. In the exercises below, unless otherwise mentioned, use the data in Figure 6.4 as the default.

a. [5] A simple way to reason about energy proportionality is to assume linearity between activity and power usage. Using just the peak power and idle power data from Figure 6.4 and a linear interpolation, plot the energy-efficiency trends across varying utilizations. (Energy efficiency is expressed as performance per watt.) What happens if idle power (at 0% activity) is half of what is assumed in Figure 6.4? What happens if idle power is zero?

b. [10] Plot the energy-efficiency trends across varying activities, but use the data from column 3 of Figure 6.4 for power variation. Plot the energy efficiency assuming that the idle power (alone) is half of what is assumed in Figure 6.4. Compare these plots with the linear model in the previous exercise. What conclusions can you draw about the consequences of focusing purely on idle power alone?

c. [15] Assume the system utilization mix in column 7 of Figure 6.4. For simplicity, assume a discrete distribution across 1000 servers, with 109 servers at 0% utilization, 80 servers at 10% utilization, etc. Compute the total performance and total energy for this workload mix using the assumptions in part (a) and part (b).

d. [20] One could potentially design a system that has a sublinear power versus load relationship in the region of load levels between 0% and 50%. This would have an energy-efficiency curve that peaks at lower utilizations (at the expense of higher utilizations). Create a new version of column 3 from Figure 6.4 that shows such an energy-efficiency curve. Assume the system utilization mix in column 7 of Figure 6.4. For simplicity, assume a discrete distribution across 1000 servers, with 109 servers at 0% utilization, 80 servers at 10% utilizations, etc. Compute the total performance and total energy for this workload mix.

6.30 [15/20/20] <6.2, 6.6> This exercise illustrates the interactions of energy proportionality models with optimizations such as server consolidation and energy-efficient server designs. Consider the scenarios shown in Figures 6.38 and 6.39.

a. [15] Consider two servers with the power distributions shown in Figure 6.38: case A (the server considered in Figure 6.4) and case B (a less energy-proportional but more energy-efficient server than case A). Assume the system utilization mix in column 7 of Figure 6.4. For simplicity, assume a discrete distribution across 1000 servers, with 109 servers at 0% utilization, 80 servers at 10% utilizations, etc., as

Activity (%)	**0**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Power, case A (W)	181	308	351	382	416	451	490	533	576	617	662
Power, case B (W)	250	275	325	340	395	405	415	425	440	445	450

Figure 6.38 Power distribution for two servers.

Activity (%)	**0**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
No. servers, case A and B	109	80	153	246	191	115	51	21	15	12	8
No. servers, case C	504	6	8	11	26	57	95	123	76	40	54

Figure 6.39 Utilization distributions across cluster, without and with consolidation.

shown in row 1 of Figure 6.39. Assume performance variation based on column 2 of Figure 6.4. Compare the total performance and total energy for this workload mix for the two server types.

b. [20] Consider a cluster of 1000 servers with data similar to the data shown in Figure 6.4 (and summarized in the first rows of Figures 6.38 and 6.39). What are the total performance and total energy for the workload mix with these assumptions? Now assume that we were able to consolidate the workloads to model the distribution shown in case C (second row of Figure 6.39). What are the total performance and total energy now? How does the total energy compare with a system that has a linear energy-proportional model with idle power of zero watts and peak power of 662 W?

c. [20] Repeat part (b), but with the power model of server B, and compare with the results of part (a).

6.31 [10/Discussion] <6.2, 6.4, 6.6> *System-Level Energy Proportionality Trends*: Consider the following breakdowns of the power consumption of a server:

CPU, 50%; memory, 23%; disks, 11%; networking/other, 16%
CPU, 33%; memory, 30%; disks, 10%; networking/other, 27%

a. [10] Assume a dynamic power range of 3.0 × for the CPU (i.e., the power consumption of the CPU at idle is one-third that of its power consumption at peak). Assume that the dynamic range of the memory systems, disks, and the networking/other categories above are, respectively, 2.0 ×, 1.3 ×, and 1.2 ×. What is the overall dynamic range for the total system for the two cases?

b. [Discussion/10] What can you learn from the results of part (a)? How would we achieve better energy proportionality at the system level? (*Hint*: Energy proportionality at a system level cannot be achieved through CPU optimizations alone, but instead requires improvement across all components.)

6.32 [30] <6.4> Pitt Turner IV et al. [2008] presented a good overview of datacenter tier classifications. Tier classifications define site infrastructure performance. For simplicity, consider the key differences as shown in Figure 6.40 (adapted from Pitt Turner IV et al. [2008]). Using the TCO model in the case study as a guiding framework, compare the cost implications of the different tiers shown.

6.33 [Discussion] <6.4> Based on the observations in Figures 6.12 and 6.13, what can you say qualitatively about the trade-offs between revenue loss from downtime and costs incurred for uptime?

6.34 [15/Discussion] <6.4> Some recent studies have defined a metric called TPUE, which stands for "true PUE" or "total PUE." TPUE is defined as PUE * SPUE. PUE, the power utilization effectiveness, is defined in Section 6.4 as the ratio of the total facility power over the total IT equipment power. SPUE, or server PUE, is a new metric analogous to PUE, but instead applied to computing equipment. SPUE is defined as the ratio of total server input power to its useful power, where useful power is defined as the power consumed by the electronic components

Tier 1	Single path for power and cooling distributions, without redundant components	99.0%
Tier 2	($N + 1$) redundancy = two power and cooling distribution paths	99.7%
Tier 3	($N + 2$) redundancy = three power and cooling distribution paths for uptime even during maintenance	99.98%
Tier 4	Two active power and cooling distribution paths, with redundant components in each path, to tolerate any single equipment failure without impacting the load	99.995%

Figure 6.40 Overview of data center tier classifications. (Adapted from Pitt Turner IV et al. [2008].).

directly involved in the computation: motherboard, disks, CPUs, DRAM, I/O cards, and so on. In other words, the SPUE metric captures inefficiencies associated with the power supplies, voltage regulators, and fans housed on a server.

a. [15] <6.4> Consider a design that uses a higher supply temperature for the computer room air conditioning (CRAC) units. The efficiency of the CRAC unit is approximately a quadratic function of the temperature, and this design therefore improves the overall PUE, let's assume by 7%. (Assume baseline PUE of 1.7.) However, the higher temperature at the server level triggers the on-board fan controller to operate the fan at much higher speeds. The fan power is a cubic function of speed, and the increased fan speed leads to a degradation of SPUE. Assume a fan power model:

$$\text{Fan power} = 284{*}ns{*}ns{*}ns - 75{*}ns{*}ns,$$

where *ns* is the normalized fan speed = fan speed in rpm/18,000 and a baseline server power of 350 W. Compute the SPUE if the fan speed increases from (1) 10,000–12,500 rpm and (2) 10,000–18,000 rpm. Compare the PUE and TPUE in both these cases. (For simplicity, ignore the inefficiencies with power delivery in the SPUE model.)

b. [Discussion] Part (a) illustrates that, while PUE is an excellent metric to capture the overhead of the facility, it does not capture the inefficiencies within the IT equipment itself. Can you identify another design where changes to the TPUE are potentially lower than the changes to traditional PUE? (*Hint*: See Exercise 6.26.)

6.35 [Discussion/30/Discussion] <6.2> Two benchmarks provide a good starting point for energy-efficiency accounting in servers—the SPECpower_ssj2008 benchmark (available at *http://www.spec.org/power_ssj2008/*) and the JouleSort metric (available at *http://sortbenchmark.org/*).

a. [Discussion] <6.2> Look up the descriptions of the two benchmarks. How are they similar? How are they different? What would you do to improve these benchmarks to better address the goal of improving WSC energy efficiency?

b. [30] <6.2> JouleSort measures the total system energy to perform an out-of-core sort and attempts to derive a metric that enables the comparison of systems ranging from embedded devices to supercomputers. Look up the description of the JouleSort metric at *http://sortbenchmark.org*. Download a publicly available version of the sort algorithm and run it on different classes of machines—a laptop, a PC, a mobile phone, etc.—or with different configurations. What can you learn from the JouleSort ratings for different setups?

c. [Discussion] <6.2> Consider the system with the best JouleSort rating from your experiments above. How would you improve the energy efficiency? For example, try rewriting the sort code to improve the JouleSort rating. What does running sort in the cloud do to energy efficiency?

6.36 [10/10/15] <6.1, 6.2> Figure 6.1 is a listing of outages in an array of servers. When dealing with the large scale of WSCs, it is important to balance cluster design and software architectures to achieve the required uptime without incurring significant costs. This question explores the implications of achieving availability through hardware only.

a. [10] <6.1, 6.2> Assuming that an operator wishes to achieve 95% availability through server hardware improvements alone, how many events of each type would have to be reduced? For now, assume that individual server crashes are completely handled through redundant machines.

b. [10] <6.1, 6.2> How does the answer to part (a) change if the individual server crashes are handled by redundancy 50% of the time? 20% of the time? None of the time?

c. [15] <6.1, 6.2> Discuss the importance of software redundancy to achieving a high level of availability. If a WSC operator considered buying machines that were cheaper but 10% less reliable, what implications would this have on the software architecture? What are the challenges associated with software redundancy?

d. [Discussion] <6.1> Discuss the importance of eventual consistency in how warehouse-scale computers can scale.

6.37 [15] <6.1, 6.8> Look up the current prices of standard DDR4 DRAM versus DDR4 DRAM that has error-correcting code (ECC). What is the increase in price per bit for achieving the higher reliability that ECC provides? Using the DRAM prices alone, and the data provided in Section 6.8, what is the uptime per dollar of a WSC with non-ECC versus ECC DRAM?

6.38 [5/Discussion] <6.1, 6.8> *WSC Reliability and Manageability Concerns*:

a. [5] Consider a cluster of servers costing $2000 each. Assuming an annual failure rate of 5%, an average of an hour of service time per repair, and replacement parts requiring 10% of the system cost per failure, what is the annual maintenance cost per server? Assume an hourly rate of $100 per hour for a service technician.

b. [Discussion] Comment on the differences between this manageability model versus that in a traditional enterprise datacenter with a large number of small- or medium-sized applications each running on its own dedicated hardware infrastructure.

c. [Discussion] Discuss the trade-offs in having heterogeneous machines in a warehouse-scale computer.

6.39 [Discussion] <6.4, 6.7, 6.8> The OpenCompute project at www.opencompute.org provides a community to design and share efficient designs for warehouse-scale computers. Look at some of the recently proposed designs. How do they compare with the design trade-offs discussed in this chapter? How do the designs differ from the Google case study discussed in Section 6.7?

6.40 [15/15] <6.3, 6.4, 6.5> Assume that the MapReduce job from Page #438 in Section 6.2 is executing a task with 2^40 bytes of input data, 2^37 bytes of intermediate data, and 2^30 bytes of output data. This job is entirely memory/storage bound, so its performance can be quantified by the DRAM/Disk bandwidth of Figure 6.6.

a. How much does the job cost to run on m4.16xlarge and m4.large in Figure 6.15? Which EC2 instance provides better performance? Which EC2 instance provides better cost?

b. How much would the job cost if an SSD was added to the system, as in m3.medium? How do the performance and cost of m3.medium compare with the best instance from part (a) above?

6.41 <6.1, 6.4> [5/5/10/Discussion] Imagine you have created a web service that runs very well (responds within 100 ms latency) 99% of the time, and has performance issues 1% of the time (maybe the CPU went into a lower power state and the response took 1000 ms, etc.).

a. [5] Your service grows popular, and you now have 100 servers and your computation has to touch all these servers to handle the user request. What is the percentage of time your query is likely to have a slow response time, across 100 servers?

b. [5] Instead of "two nines" (99%) single server latency SLA, how many "nines" do we need to have for the single server latency SLA so that the cluster latency SLA has bad latencies only 10% of the time or lower?

c. [10] How do the answers to parts (a) and (b) change if we have 2000 servers?

d. [Discussion] Section 6.4 (page 452) discusses "tail-tolerant" designs. What kind of design optimizations would you need to make in your web service (*Hint*: Look at the "Tail at Scale" paper from Dean and Barroso [2013]).

7.1 Introduction 540
7.2 Guidelines for DSAs 543
7.3 Example Domain: Deep Neural Networks 544
7.4 Google’s Tensor Processing Unit, an Inference Data Center Accelerator 557
7.5 Microsoft Catapult, a Flexible Data Center Accelerator 567
7.6 Intel Crest, a Data Center Accelerator for Training 579
7.7 Pixel Visual Core, a Personal Mobile Device Image Processing Unit 579
7.8 Cross-Cutting Issues 592
7.9 Putting It All Together: CPUs Versus GPUs Versus DNN Accelerators 595
7.10 Fallacies and Pitfalls 602
7.11 Concluding Remarks 604
7.12 Historical Perspectives and References 606
Case Studies and Exercises by Cliff Young 606

7

Domain-Specific Architectures

Moore's Law can't continue forever … We have another 10 to 20 years before we reach a fundamental limit

Gordon Moore,
Intel Co-Founder (2005)

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00007-9
© 2019 Elsevier Inc. All rights reserved.

7.1 Introduction

Gordon Moore not only predicted the amazing growth of transistors per chip in 1965, but the opening chapter quote shows that he also predicted its demise 50 years later. As evidence, Figure 7.1 shows that even the company he founded—which for decades proudly used Moore's Law as a guideline for capital investment—is slowing its development of new semiconductor processes.

During the semiconductor boom time, architects rode Moore's Law to create novel mechanisms that could turn the cornucopia of transistors into higher performance. The resources for a five-stage pipeline, 32-bit RISC processor—which needed as little as 25,000 transistors in the 1980s—grew by a factor of 100,000 to enable features that accelerated general-purpose code on general-purpose processors, as earlier chapters document:

- 1st-level, 2nd-level, 3rd-level, and even 4th-level caches
- 512-bit SIMD floating-point units
- 15+ stage pipelines
- Branch prediction
- Out-of-order execution
- Speculative prefetching
- Multithreading
- Multiprocessing

These sophisticated architectures targeted million-line programs written in efficient languages like C++. Architects treated such code as black boxes, generally

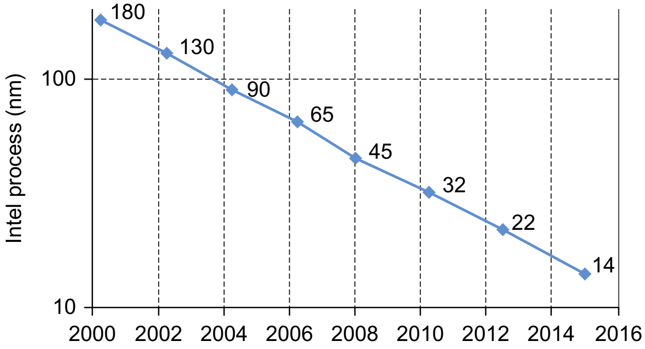


Figure 7.1 Time before new Intel semiconductor process technology measured in nm. The *y*-axis is log scale. Note that the time stretched previously from about 24 months per new process step to about 30 months since 2010.

without understanding either the internal structure of the programs or even what they were trying to do. Benchmark programs like those in SPEC2017 were just artifacts to measure and accelerate. Compiler writers were the people at the hardware-software interface, which dates back to the RISC revolution in the 1980s, but they have limited understanding of the high-level application behavior; that's why compilers cannot even bridge the semantic gap between C or C++ and the architecture of GPUs.

As Chapter 1 described, Dennard scaling ended much earlier than Moore's Law. Thus more transistors switching now means more power. The energy budget is not increasing, and we've already replaced the single inefficient processor with multiple efficient cores. Hence, we have nothing left up our sleeves to continue major improvements in cost-performance and energy efficiency for general-purpose architectures. Because the energy budget is limited (because of electromigration, mechanical and thermal limits of chips), if we want higher performance (higher operations/second), we need to lower the energy per operation.

Figure 7.2 is another take on the relative energy costs of memory and logic mentioned in Chapter 1, this time calculated as overhead for an arithmetic instruction. Given this overhead, minor twists to existing cores may get us 10% improvements, but if we want order-of-magnitude improvements while offering programmability, we need to increase the number of arithmetic operations per instruction from one to hundreds. To achieve that level of efficiency, we need a drastic change in computer architecture from general-purpose cores to *domain-specific architectures (DSAs)*.

Thus, just as the field switched from uniprocessors to multiprocessors in the past decade out of necessity, desperation is the reason architects are now working on DSAs. The new normal is that a computer will consist of standard processors to run conventional large programs such as operating systems along with domain-specific processors that do only a narrow range of tasks, but they do them extremely well. Thus such computers will be much more heterogeneous than the homogeneous multicore chips of the past.

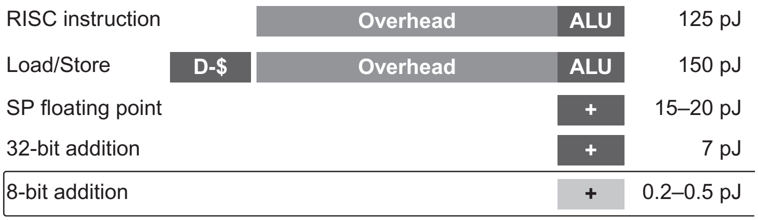


Figure 7.2 Energy costs in picoJoules for a 90 nm process to fetch instructions or access a data cache compared to the energy cost of arithmetic operations (Qadeer et al., 2015).

Part of the argument is that the preceding architecture innovations from the past few decades that leveraged Moore's Law (caches, out-of-order execution, etc.) may not be a good match to some domains—especially in terms of energy usage—so their resources can be recycled to make the chip a better match to the domain. For example, caches are excellent for general-purpose architectures, but not necessarily for DSAs; for applications with easily predictable memory access patterns or huge data sets like video that have little data reuse, multilevel caches are overkill, hording area and energy that could be put to better use. Therefore the promise of DSAs is both improved silicon efficiency and better energy efficiency, with the latter typically being the more important attribute today.

Architects probably won't create a DSA for a large C++ program like a compiler as found in the SPEC2017 benchmark. Domain-specific algorithms are almost always for small compute-intensive kernels of larger systems, such as for object recognition or speech understanding. DSAs should focus on the subset and not plan to run the entire program. In addition, changing the code of the benchmark is no longer breaking the rules; it is a perfectly valid source of speedup for DSAs. Consequently, if they are going to make useful contributions, architects interested in DSA must now shed their blinders and learn application domains and algorithms.

In addition to needing to expand their areas of expertise, a challenge for domain-specific architects is to find a target whose demand is large enough to justify allocating dedicated silicon on an SOC or even a custom chip. The *nonrecurring engineering (NRE)* costs of a custom chip and supporting software are amortized over the number of chips manufactured, so it is unlikely to make economic sense if you need only 1000 chips.

One way to accommodate smaller volume applications is to use reconfigurable chips such as FPGAs because they have lower NRE than custom chips *and* because several different applications may be able to reuse the same reconfigurable hardware to amortize its costs (see Section 7.5). However, since the hardware is less efficient than custom chips, the gains from FPGAs are more modest.

Another DSA challenge is how to port software to it. Familiar programming environments like the C++ programming language and compiler are rarely the right vehicles for a DSA.

The rest of this chapter provides five guidelines for the design of DSAs and then a tutorial on our example domain, which is *deep neural networks* (*DNNs*). We chose DNNs because they are revolutionizing many areas of computing today. Unlike some hardware targets, DNNs are applicable to a wide range of problems, so we can reuse a DNN-specific architecture for solutions in speech, vision, language, translation, search ranking, and many more areas.

We follow with four examples of DSAs: two custom chips for the data center that accelerate DNNs, an FPGA for the data center that accelerates many domains, and an image-processing unit designed for *personal mobile devices* (*PMDs*). We then compare the cost-performance of the DSAs along with CPUs and GPUs using DNN benchmarks, and conclude with a prediction of an upcoming renaissance for computer architecture.

7.2 Guidelines for DSAs

Here are five principles that generally guided the designs of the four DSAs we'll see in Sections 7.4–7.7. Not only do these five guidelines lead to increased area and energy efficiency, they also provide two valuable bonus effects. First, they lead to simpler designs, which reduce the cost of NRE of DSAs (see the fallacy in Section 7.10). Second, for user-facing applications that are commonplace with DSAs, accelerators that follow these principles are a better match to the 99th-percentile response-time deadlines than the time-varying performance optimizations of traditional processors, as we will see in Section 7.9. Figure 7.3 shows how the four DSAs followed these guidelines.

1. *Use dedicated memories to minimize the distance over which data is moved.* The many levels of caches in general-purpose microprocessors use a great deal of area and energy trying to move data optimally for a program. For example, a two-way set associative cache uses 2.5 times as much energy as an equivalent software-controlled scratchpad memory. By definition, the compiler writers and programmers of DSAs understand their domain, so there is no need for the hardware to try to move data for them. Instead, data movement is reduced with software-controlled memories that are dedicated to and tailored for specific functions within the domain.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.* As Section 7.1 describes, architects turned the bounty from Moore's Law into the resource-intensive optimizations for CPUs and GPUs (out-of-order execution, multithreading, multiprocessing, prefetching, address coalescing, etc.).

Guideline	TPU	Catapult	Crest	Pixel Visual Core
Design target	Data center ASIC	Data center FPGA	Data center ASIC	PMD ASIC/SOC IP
1. Dedicated memories	24 MiB Unified Buffer, 4 MiB Accumulators	Varies	N.A.	Per core: 128 KiB line buffer, 64 KiB P.E. memory
2. Larger arithmetic unit	65,536 Multiply-accumulators	Varies	N.A.	Per core: 256 Multiply-accumulators (512 ALUs)
3. Easy parallelism	Single-threaded, SIMD, in-order	SIMD, MISD	N.A.	MPMD, SIMD, VLIW
4. Smaller data size	8-Bit, 16-bit integer	8-Bit, 16-bit integer 32-bit Fl. Pt.	21-bit Fl. Pt.	8-bit, 16-bit, 32-bit integer
5. Domain-specific lang.	TensorFlow	Verilog	TensorFlow	Halide/TensorFlow

Figure 7.3 The four DSAs in this chapter and how closely they followed the five guidelines. Pixel Visual Core typically has 2–16 cores. The first implementation of Pixel Visual Core does not support 8-bit arithmetic.

Given the superior understanding of the execution of programs in these narrower domains, these resources are much better spent on more processing units or larger on-chip memory.

3. *Use the easiest form of parallelism that matches the domain.*
   Target domains for DSAs almost always have inherent parallelism. The key decisions for a DSA are how to take advantage of that parallelism and how to expose it to the software. Design the DSA around the natural granularity of the parallelism of the domain and expose that parallelism simply in the programming model. For example, with respect to data-level parallelism, if SIMD works in the domain, it's certainly easier for the programmer and the compiler writer than MIMD. Similarly, if VLIW can express the instruction-level parallelism for the domain, the design can be smaller and more energy-efficient than out-of-order execution.
4. *Reduce data size and type to the simplest needed for the domain.*
   As we will see, applications in many domains are typically memory-bound, so you can increase the effective memory bandwidth and on-chip memory utilization by using narrower data types. Narrower and simpler data also let's you pack more arithmetic units into the same chip area.
5. *Use a domain-specific programming language to port code to the DSA.*
   As Section 7.1 mentions, a classic challenge for DSAs is getting applications to run on your novel architecture. A long-standing fallacy is assuming that your new computer is so attractive that programmers will rewrite their code just for your hardware. Fortunately, domain-specific programming languages were becoming popular even before architects were forced to switch their attention to DSAs. Examples are Halide for vision processing and TensorFlow for DNNs (Ragan-Kelley et al., 2013; Abadi et al., 2016). Such languages make porting applications to your DSA much more feasible. As previously mentioned, only a small, compute-intensive portion of the application needs to run on the DSA in some domains, which also simplifies porting.

DSAs introduce many new terms, mostly from the new domains but also from novel architecture mechanisms not seen in conventional processors. As we did in Chapter 4, Figure 7.4 lists the new acronyms, terms, and short explanations to aid the reader.

7.3 Example Domain: Deep Neural Networks

> Artificial intelligence (AI) is not only the next big wave in computing—it's the next major turning point in human history… the Intelligence Revolution will be driven by data, neural networks and computing power. Intel is committed to AI [thus]… we've added a set of leading-edge accelerants required for the growth and widespread adoption of AI.
>
> **Brian Krzanich**,
> *Intel CEO* (2016)

Area	Term	Acronym	Short explanation
General	Domain-specific architectures	DSA	A special-purpose processor designed for a particular domain. It relies on other processors to handle processing outside that domain
	Intellectual property block	IP	A portable design block that can be integrated into an SOC. They enable a marketplace where organizations offer IP blocks to others who compose them into SOCs
	System on a chip	SOC	A chip that integrates all the components of a computer; commonly found in PMDs
Deep neural networks	Activation	—	Result of "activating" the artificial neuron; the output of the nonlinear functions
	Batch	—	A collection of datasets processed together to lower the cost of fetching weights
	Convolutional neural network	CNN	A DNN that takes as inputs a set of nonlinear functions of spatially nearby regions of outputs from the prior layer, which are multiplied by the weights
	Deep neural network	DNN	A sequence of layers that are collections of artificial neurons, which consist of a nonlinear function applied to products of weights times the outputs of the prior layer
	Inference	—	The production phase of DNNs; also called *prediction*
	Long short-term memory	LSTM	An RNN well suited to classify, process, and predict time series. It is a hierarchical design consisting of modules called *cells*
	MultiLayer perceptron	MLP	A DNN that takes as inputs a set of nonlinear functions of all outputs from the prior layer multiplied by the weights. These layers are called *fully connected*
	Rectified linear unit	*ReLU*	A nonlinear function that performs `f(x)=max(x,0)`. Other popular nonlinear functions are sigmoid and hyperbolic tangent (tanh)
	Recurrent neural network	RNN	A DNN whose inputs are from the prior layer *and* the previous state
	Training	—	The development phase of DNNs; also called *learning*
	Weights	—	The values learned during training that are applied to inputs; also called *parameters*
TPU	Accumulators	—	The 4096 256 × 32-bit registers (4 MiB) that collect the output of the MMU and are input to the Activation Unit
	Activation unit	—	Performs the nonlinear functions (ReLU, sigmoid, hyperbolic tangent, max pool, and average pool). Its input comes from the Accumulators and its output goes to the Unified Buffer
	Matrix multiply unit	MMU	A systolic array of 256 × 256 8-bit arithmetic units that perform multiply-add. Its inputs are the Weight Memory and the Unified Buffer, and its output is the Accumulators
	Systolic array	—	An array of processing units that in lockstep input data from upstream neighbors, compute partial results, and pass some inputs and results to downstream neighbors
	Unified buffer	UB	A 24 MiB on-chip memory that holds the activations. It was sized to try to avoid spilling activations to DRAM when running a DNN
	Weight memory	—	An 8 MiB external DRAM chip containing the weights for the MMU. Weights are transferred to a *Weight FIFO* before entering the MMU

Figure 7.4 A handy guide to DSA terms used in Sections 7.3–7.6. Figure 7.29 on page 472 has a guide for Section 7.7.

Artificial intelligence (AI) has made a dramatic comeback since the turn of the century. Instead of *building* artificial intelligence as a large set of logical rules, the focus switched to *machine learning* from example data as the path to artificial intelligence. The amount of data needed to learn was much greater than thought. The warehouse scale computers (WSCs) of this century, which harvest and store petabytes of information found on the Internet from the billions of users and their smartphones, supply the ample data. We also underestimated the amount of computation needed to learn from the massive data, but GPUs—which have excellent single-precision floating-point cost-performance—embedded in the thousands of servers of WSCs deliver sufficient computing.

One part of machine learning, called DNNs, has been the AI star for the past five years. Example DNN breakthroughs are in language translation, which DNNs improved more in a single leap than all the advances from the prior decade (Tung, 2016; Lewis-Kraus, 2016); the switch to DNNs in the past five years reduced the error rate in an image recognition competition from 26% to 3.5% (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016); and in 2016, DNNs enabled a computer program for the first time to beat a human champion at Go (Silver et al., 2016). Although many of these run in the cloud, they have also enabled Google Translate on smartphones, which we described in Chapter 1. In 2017 new, significant DNN results appear nearly every week.

Readers interested in learning more about DNNs than found in this section should download and try the tutorials in TensorFlow (TensorFlow Tutorials, 2016), or for the less adventurous, consult a free online textbook on DNNs (Nielsen, 2016).

The Neurons of DNNs

DNNs were inspired by the neuron of the brain. The artificial neuron used for neural networks simply computes the sum over a set of products of *weights* or *parameters* and data values that is then put through a nonlinear function to determine its output. As we will see, each artificial neuron has a large fan-in and a large fan-out.

For an image-processing DNN, the input data would be the pixels of a photo, with the pixel values multiplied by the weights. Although many nonlinear functions have been tried, a popular one today is simply `f(x)=max(x,0)`, which returns 0 if the `x` is negative or the original value if positive or zero. (This simple function goes by the complicated name *rectified linear unit* or *ReLU*.) The output of a nonlinear function is called an *activation*, in that it is the output of the artificial neuron that has been "activated."

A cluster of artificial neurons might process different portions of the input, and the output of that cluster becomes the input to the next layer of artificial neurons. The layers between the input layer and the output layer are called *hidden layers*. For image processing, you can think of each layer as looking for different types of features, going from lower-level ones like edges and angles to higher-level ones like eyes and ears. If the image-processing application was trying to decide if the image

Name	DNN layers	Weights	Operations/Weight
MLP0	5	20M	200
MLP1	4	5M	168
LSTM0	58	52M	64
LSTM1	56	34M	96
CNN0	16	8M	2888
CNN1	89	100M	1750

Figure 7.5 Six DNN applications that represent 95% of DNN workloads for inference at Google in 2016, which we use in Section 7.9. The columns are the DNN name, the number of layers in the DNN, the number of weights, and operations per weight (operational intensity). Figure 7.41 on page 595 goes into more detail on these DNNs.

contained a dog, the output of the last layer could be a probability number between 0 and 1 or perhaps a list of probabilities corresponding to a list of dog breeds.

The number of layers gave DNNs their name. The original lack of data and computing horsepower kept most neural networks relatively shallow. Figure 7.5 shows the number of layers for a variety of recent DNNs, the number of weights, and the number of operations per weight fetched. In 2017 some DNNs have 150 layers.

Training Versus Inference

The preceding discussion concerns DNNs that are in production. DNN development starts by defining the neural network architecture, picking the number and type of layers, the dimensions of each layer, and the size of the data. Although experts may develop new neural network architectures, most practitioners will choose among the many existing designs (e.g., Figure 7.5) that have been shown to perform well on problems similar to theirs.

Once the neural architecture has been selected, the next step is to learn the weights associated with each edge in the neural network graph. The weights determine the behavior of the model. Depending on the choice of neural architecture, there can be anywhere from thousands to hundreds of millions of weights in a single model (see Figure 7.5). Training is the costly process of tuning these weights so that the DNN approximates the complex function (e.g., mapping from pictures to the objects in that picture) described by the training data.

This development phase is universally called *training* or *learning*, whereas the production phase has many names: *inference*, *prediction*, *scoring*, *implementation*, *evaluation*, *running*, or *testing*. Most DNNs use *supervised learning* in that they are given a training set to learn from where the data is preprocessed in order to have the correct labels. Thus, in the ImageNet DNN competition (Russakovsky et al., 2015), the training set consists of 1.2 million photos, and each photo has been labeled as one of 1000 categories. Several of these categories are quite detailed, such as specific breeds of dogs and cats. The winner is determined by evaluating a separate secret set of 50,000 photos to see which DNN has the lowest error rate.

Setting the weights is an iterative process that goes *backward* through the neural network using the training set. This process is called *backpropagation*. For example, because you know the breed of a dog image in the training set, you see what your DNN says about the image, and then you adjust the weights to improve the answer. Amazingly, the weights at the start of the training process should be set to random data, and you just keep iterating until you're satisfied with the DNN accuracy using the training set.

For the mathematically inclined, the goal of learning is to find a function that maps the inputs to the correct outputs over the multilayer neural network architecture. Backpropagation stands for "back propagation of errors." It calculates a gradient over all the weights as input to an optimization algorithm that tries to minimize the errors by updating the weights. The most popular optimization algorithm for DNNs is *stochastic gradient descent*. It adjusts the weights proportionally to maximize the descent of the gradient obtained from backpropagation. Readers interested in learning more should see Nielsen (2016) or TensorFlow Tutorials (2016).

Training can take weeks of computation, as Figure 7.6 shows. The inference phase is often below 100 ms per data sample, which is a million times less. Although training takes much longer than a single inference, the total compute time for inference is a product of the number of customers of the DNN and how frequently they invoke it.

After training, you deploy your DNN, hoping that your training set is representative of the real world, and that your DNN will be so popular that your users will spend much more time employing it than you've put into developing it!

Type of data	Problem area	Size of benchmark's training set	DNN architecture	Hardware	Training time
text [1]	Word prediction (word2vec)	100 billion words (Wikipedia)	2-layer skip gram	1 NVIDIA Titan X GPU	6.2 hours
audio [2]	Speech recognition	2000 hours (Fisher Corpus)	11-layer RNN	1 NVIDIA K1200 GPU	3.5 days
images [3]	Image classification	1 million images (ImageNet)	22-layer CNN	1 NVIDIA K20 GPU	3 weeks
video [4]	activity recognition	1 million videos (Sports-1M)	8-layer CNN	10 NVIDIA GPUs	1 month

Figure 7.6 Training set sizes and training time for several DNNs (Iandola, 2016).

There are tasks that don't have training datasets, such as when trying to predict the future of some real-world event. Although we won't cover it here, *reinforcement learning* (*RL*) is a popular algorithm for such learning in 2017. Instead of a training set to learn from, RL acts on the real world and then gets a signal from a reward function, depending on whether that action made the situation better or worse.

Although it's hard to imagine a faster changing field, only three types of DNNs reign as most popular in 2017: *MultiLayer Perceptrons* (*MLPs*), *Convolutional Neural Networks* (*CNNs*), and *Recurrent Neural Networks* (*RNNs*). They are all examples of supervised learning, which rely on training sets.

Multilayer Perceptron

MLPs were the original DNNs. Each new layer is a set of nonlinear functions F of weighted sum of all outputs from a prior one $y_n = F(W \times y_{n-1})$. The weighted sum consists of a vector-matrix multiply of the outputs with the weights (see Figure 7.7). Such a layer is called *fully connected* because each output neuron result depends on *all* input neurons of the prior layer.

We can calculate the number of neurons, operations, and weights per layer for each of the DNN types. The easiest is MLP because it is just a vector-matrix

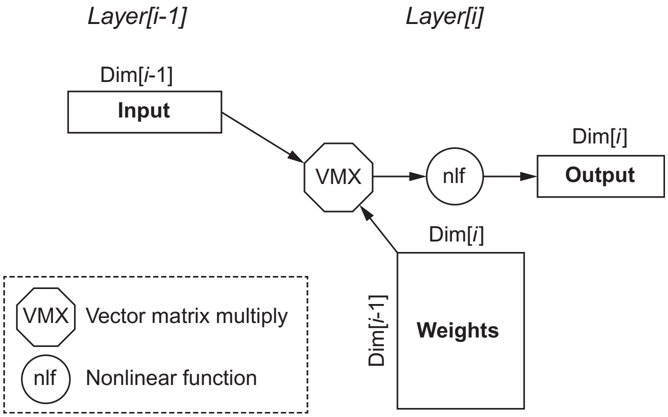


Figure 7.7 MLP showing the input Layer[*i*−1] on the left and the output Layer[*i*] on the right. ReLU is a popular nonlinear function for MLPs. The dimensions of the input and output layers are often different. Such a layer is called fully connected because it depends on all the inputs from the prior layer, even if many of them are zeros. One study suggested that 44% were zeros, which presumably is in part because ReLU turns negative numbers into zeros.

multiply of the input vector times the weights array. Here are the parameters and the equations to determine weights and operations for inference (we count multiply and add as two operations):

- Dim[i]: Dimension of the output vector, which is the number of neurons
- Dim[$i-1$]: Dimension of the input vector
- Number of weights: Dim[$i-1$] × Dim[i]
- Operations: 2 × Number of weights
- Operations/Weight: 2

This final term is the *operational intensity* from the Roofline model discussed in Chapter 4. We use operations per *weight* because there can be millions of weights, which usually don't fit on the chip. For example, the dimensions of one stage of an MLP in Section 7.9 has Dim[$i-1$] = 4096 and Dim[i] = 2048, so for that layer, the number of neurons is 2048, number of weights is 8,388,608, the number of operations is 16,777,216, and the operational intensity is 2. As we recall from the Roofline model, low operational intensity makes it harder to deliver high performance.

Convolutional Neural Network

CNNs are widely used for computer vision applications. As images have a two-dimensional structure, neighboring pixels are the natural place to look to find relationships. CNNs take as inputs a set of nonlinear functions from spatially nearby regions of outputs from the prior layer and then multiplies by the weights, which reuses the weights many times.

The idea behind CNNs is that each layer raises the level of abstraction of the image. For example, the first layer might identify only horizontal lines and vertical lines. The second layer might combine them to identify corners. The next step might be rectangles and circles. The following layer could use that input to detect portions of a dog, like eyes or ears. The higher layers would be trying to identify characteristics of different breeds of dogs.

Each neural layer produces a set of two-dimensional *feature maps*, where each cell of the two-dimensional feature map is trying to identify one feature in the corresponding area of the input.

Figure 7.8 shows the starting point where a 2 × 2 stencil computation from the input image creates the elements of the first feature map. A *stencil computation* uses neighboring cells in a fixed pattern to update all the elements of an array. The number of output feature maps will depend on how many different features you are trying to capture from the image and the stride used to apply the stencil.

The process is actually more complicated because the image is usually not just a single, flat two-dimensional layer. Typically, a color image will have three levels for red, green, and blue. For example, a 2 × 2 stencil will access 12 elements: 2 × 2

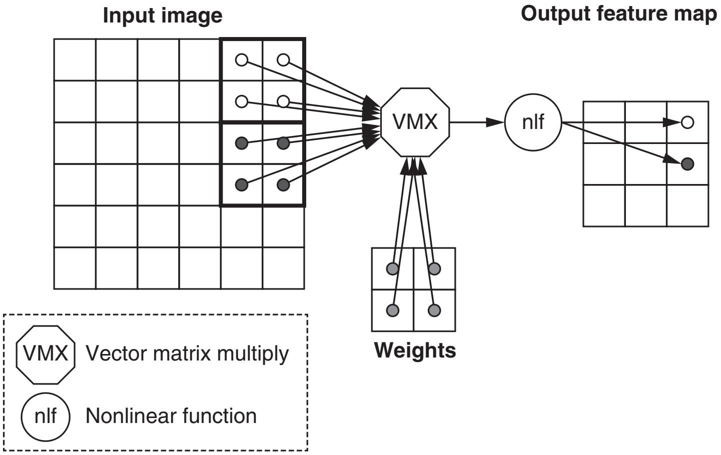


Figure 7.8 Simplified first step of a CNN. In this example, every group of four pixels of the input image are multiplied by the same four weights to create the cells of the output feature map. The pattern depicted shows a stride of two between the groups of input pixels, but other strides are possible. To relate this figure to MLP, you can think of each 2×2 convolution as a tiny fully connected operation to produce one point of the output feature map. Figure 7.9 shows how multiple feature maps turn the points into a vector in the third dimension.

of red pixels, 2×2 of green pixels, and 2×2 of blue pixels. In this case, you need 12 weights per output feature map for a 2×2 stencil on three input levels of an image.

Figure 7.9 shows the general case of an arbitrary number of input and output feature maps, which occurs after that first layer. The calculation is a three-dimensional stencil over all the input feature maps with a set of weights to produce one output feature map.

For the mathematically oriented, if the number of input feature maps and output feature maps both equal 1 and the stride is 1, then a single layer of a two-dimensional CNN is the same calculation as a two-dimensional discrete convolution.

As we see in Figure 7.9, CNNs are more complicated than MLPs. Here are the parameter and the equations to calculate the weights and operations:

- DimFM[$i-1$]: Dimension of the (square) input Feature Map
- DimFM[i]: Dimension of the (square) output Feature Map
- DimSten[i]: Dimension of the (square) stencil
- NumFM[$i-1$]: Number of input Feature Maps
- NumFM[i]: Number of output Feature Maps
- Number of neurons: NumFM[i] $\times$ DimFM[i]^2

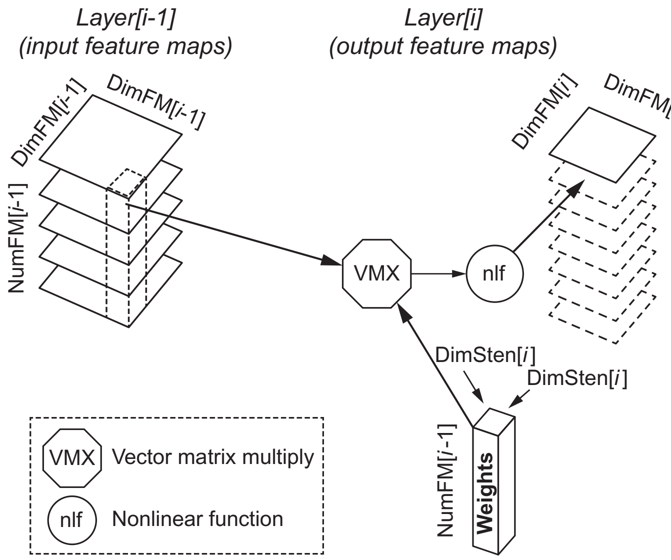

Layer[i-1]
(input feature maps)
Layer[i]
(output feature maps)
DimFM[i-1]
DimFM[i-1]
NumFM[i-1]
DimFM[i]
DimFM[i]
NumFM[i]
VMX
nlf
DimSten[i]
DimSten[i]
NumFM[i-1]
Weights
VMX Vector matrix multiply
nlf Nonlinear function

Figure 7.9 CNN general step showing input feature maps of Layer[*i*−1] on the left, the output feature maps of Layer[*i*] on the right, and a three-dimensional stencil over input feature maps to produce a single output feature map. Each output feature map has its own unique set of weights, and the vector-matrix multiply happens for every one. The dotted lines show future output feature maps in this figure. As this figure illustrates, the dimensions and number of the input and output feature maps are often different. As with MLPs, ReLU is a popular nonlinear function for CNNs.

- Number of weights per output Feature Map: NumFM[$i-1$] × DimSten[i]^2
- Total number of weights per layer: NumFM[i] × Number of weights per output Feature Map
- Number of operations per output Feature Map: 2 × DimFM[i]^2 × Number of weights per output Feature Map
- Total number of operations per layer: NumFM[i] × Number of operations per output Feature Map = 2 × DimFM[i]^2 × NumFM[i] × Number of weights per output Feature Map = 2 × DimFM[i]^2 × Total number of weights per layer
- Operations/Weight: 2 × DimFM[i]^2

A CNN in Section 7.9 has a layer with DimFM[$i-1$] = 28, DimFM[i] = 14, DimSten[i] = 3, NumFM[$i-1$] = 64 (number of input feature maps), and NumFM[i] = 128 (number of output feature maps). That layer has 25,088 neurons, 73,728 weights, does 28,901,376 operations, and has an operational intensity of 392. As our example indicates, CNN layers generally have fewer weights and greater operational intensity than the fully connected layers found in MLPs.

Recurrent Neural Network

The third type of DNN is RNNs, which are popular for speech recognition or language translation. RNNs add the ability to explicitly model sequential inputs by adding state to the DNN model so that RNNs can remember facts. It's analogous to the difference in hardware between combinational logic and a state machine. For example, you might learn the gender of the person, which you would want to pass along to remember later when translating words. Each layer of an RNN is a collection of weighted sums of inputs from the prior layer and the previous state. The weights are reused across time steps.

Long short-term memory (*LSTM*) is by far the most popular RNN today. LSTMs mitigate a problem that previous RNNs had with their inability to remember important long-term information.

Unlike the other two DNNs, LSTM is a hierarchical design. LSTM consists of modules called *cells*. You can think of cells as templates or macros that are linked together to create the full DNN model, similar to how layers of an MLP line up to form a complete DNN model.

Figure 7.10 shows how the LSTM cells are linked together. They are hooked up from left to right, connecting the output of one cell to the input of the next. They are also unrolled in time, which runs top down in Figure 7.10. Thus a sentence is input a word at a time per iteration of the unrolled loop. The long-term and short-term memory information that gives the LSTM its name is also passed top-down from one iteration to the next.

Figure 7.11 shows the contents of an LSTM cell. As we would expect from Figure 7.10, the input is on the left, the output is on the right, the two memory inputs are at the top, and the two memory outputs are at the bottom.

Each cell does five vector-matrix multiplies using five unique sets of weights. The matrix multiply on the input is just like the MLP in Figure 7.7. Three others are called *gates* in that they gate or limit how much information from one source is passed along to the standard output or the memory output. The amount of information sent per gate is set by their weights. If the weights are mostly zeros or small values, then little gets through; conversely, if they are mostly large, then the gate lets most information flow. The three gates are called the *input gate*, the *output gate*, and the *forget gate*. The first two filter the input and output, and the last one determines what to forget along the long-term memory path.

The short-term memory output is a vector-matrix multiply using the Short Term Weights and the output of this cell. The short-term label is applied because it does not directly use any of the inputs to the cell.

Because the LSTM cell inputs and outputs are all connected together, the size of the three input-output pairs must be the same. Looking inside the cell, there are enough dependencies that all of the inputs and outputs are often the same size. Let's assume they are all the same size, called *Dim*.

Even so, the vector-matrix multiplies are not all the same size. The vectors for the three gate multiplies are $3 \times \mathrm{Dim}$, because the LSTM concatenates all three inputs. The vector for the input multiply is $2 \times \mathrm{Dim}$, because the LSTM

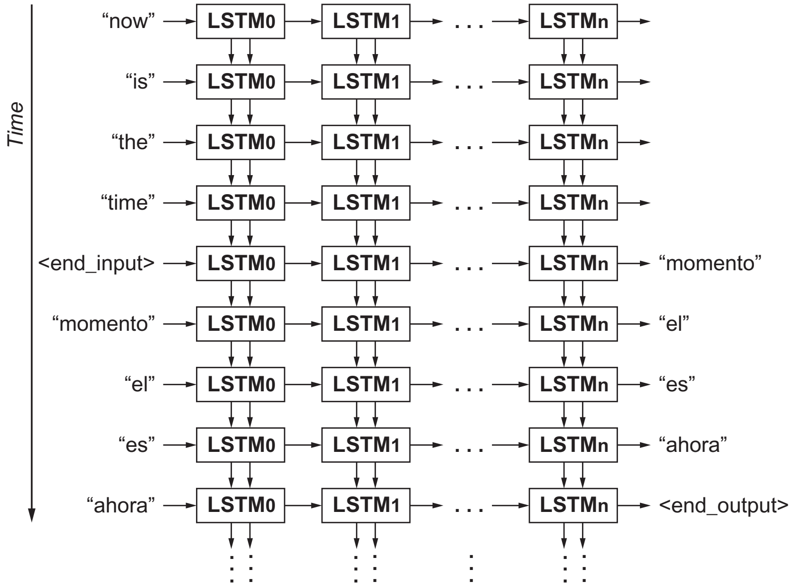


Figure 7.10 LSTM cells connected together. The inputs are on the left (English words), and the outputs are on the right (the translated Spanish words). The cells can be thought of as being unrolled over time, from top to bottom. Thus the short-term and long-term memory of LSTM is implemented by passing information top-down between unrolled cells. They are unrolled enough to translate whole sentences or even paragraphs. Such sequence-to-sequence translation models delay their output until they get to the end of the input (Wu et al., 2016). They produce the translation in *reverse order*, using the most recent translated word as input to the next step, so "now is the time" becomes "ahora es el momento." (This figure and the next are often shown turned 90 degrees in LSTM literature, but we've rotated them to be consistent with Figures 7.7 and 7.8.)

concatenates the input with the short-term memory input as the vector. The vector for the last multiply is just $1 \times \text{Dim}$, because it is just the output.

Now we can finally calculate the weights and operations:

- Number of weights per cell: $3 \times (3 \times \text{Dim} \times \text{Dim}) + (2 \times \text{Dim} \times \text{Dim}) + (1 \times \text{Dim} \times \text{Dim}) = 12 \times \text{Dim}^2$
- Number of operations for the 5 vector-matrix multiplies per cell: $2 \times$ Number of weights per cell $= 24 \times \text{Dim}^2$
- Number of operations for the 3 element-wise multiplies and 1 addition (vectors are all the size of the output): $4 \times \text{Dim}$
- Total number of operations per cell (5 vector-matrix multiplies and the 4 element-wise operations): $24 \times \text{Dim}^2 + 4 \times \text{Dim}$
- Operations/Weight: ~ 2

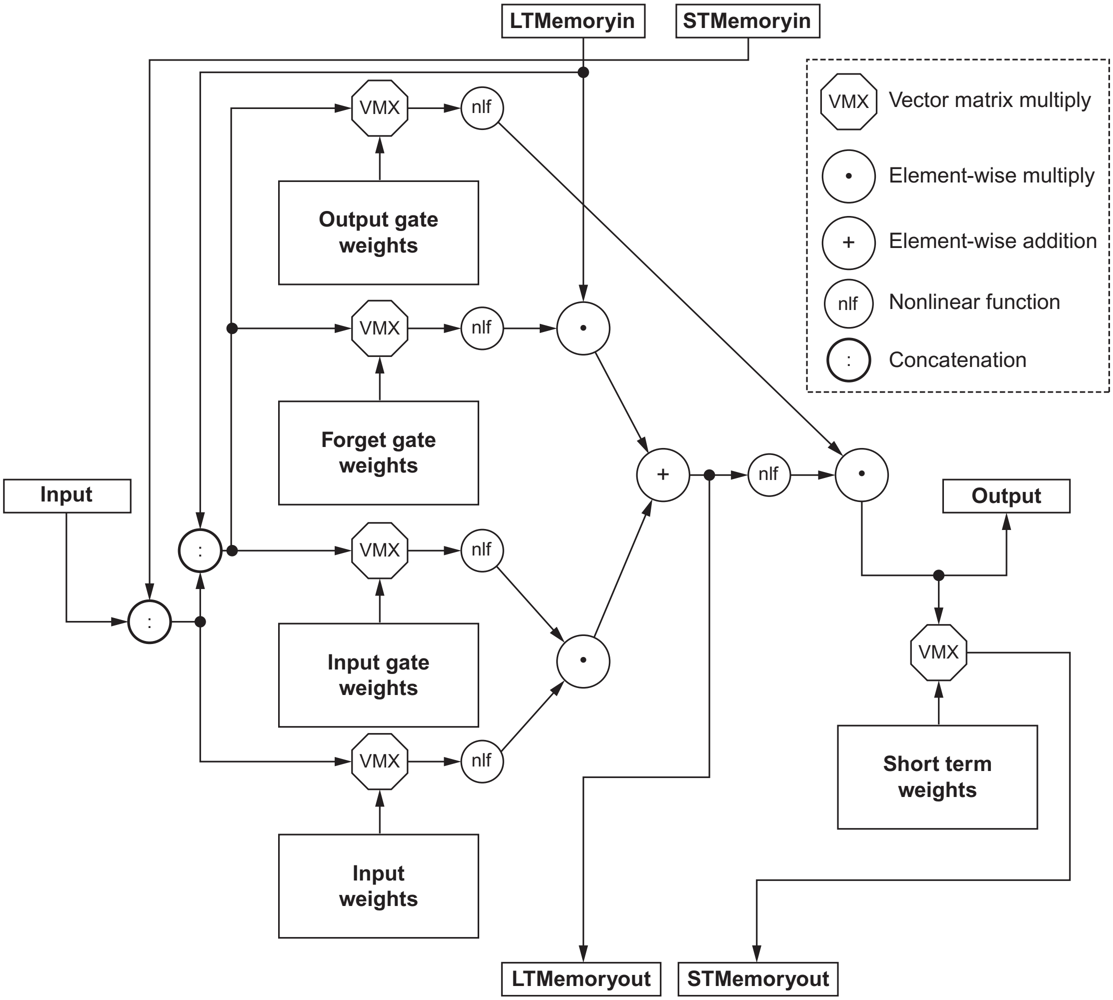

LTMemoryin
STMemoryin
VMX Vector matrix multiply
• Element-wise multiply
+ Element-wise addition
nlf Nonlinear function
: Concatenation
VMX
nlf
Output gate weights
VMX
nlf
•
Forget gate weights
+
nlf
•
Input
:
VMX
nlf
:
Output
Input gate weights
•
VMX
Short term weights
VMX
nlf
Input weights
LTMemoryout
STMemoryout

Figure 7.11 This LSTM cell contains 5 vector-matrix multiplies, 3 element-wise multiplies, 1 element-wise add, and 6 nonlinear functions. The standard input and short-term memory input are concatenated to form the vector operand for the input vector-matrix multiply. The standard input, long-term memory input, and short-term memory input are concatenated to form the vector that is used in three of the other four vector-matrix multiplies. The nonlinear functions for the three gates are Sigmoids $f(x) = 1/(1 + \exp(-x))$; the others are hyperbolic tangents. (This figure and the previous one are often shown turned 90 degrees in LSTM literature, but we've rotated them to be consistent with Figures 7.7 and 7.8.)

Dim is 1024 for one of the six cells of an LSTM in Section 7.9. Its number of weights is 12,582,912, its number of operations is 25,169,920, and its operational intensity is 2.0003. Thus LSTMs are like MLPs in that they typically have more weights and a lower operational intensity than CNNs.

Batches

Because DNNs can have many weights, a performance optimization is to reuse the weights once they have been fetched from memory across a set of inputs, thereby increasing effective operational intensity. For example, an image-processing DNN might work on a set of 32 images at a time to reduce the effective cost of fetching weights by a factor of 32. Such datasets are called *batches* or *minibatches*. In addition to improving the performance of inference, backpropagation needs a batch of examples instead of one at a time in order to train well.

Looking at an MLP in Figure 7.7, a batch can be seen as a sequence of input row vectors, which you can think of as a matrix with a height dimension that matches the batch size. A sequence of row vector inputs to the five matrix multiplies of LSTMs in Figure 7.11 can also be considered a matrix. In both cases, computing them as matrices instead of sequentially as independent vectors improves computing efficiency.

Quantization

Numerical precision is less important for DNNs than for many applications. For example, there is no need for double-precision floating-point arithmetic, which is the standard bearer of high-performance computing. It's even unclear that you need the full accuracy of the IEEE 754 floating-point standard, which aims to be accurate within one-half of a unit in the last place of the floating-point significand.

To take advantage of the flexibility in numerical precision, some developers use fixed point instead of floating point for the inference phase. (Training is almost always done in floating-point arithmetic.) This conversion is called *quantization*, and such a transformed application is said to be *quantized* (Vanhoucke et al., 2011). The fixed-point data width is usually 8 or 16 bits, with the standard multiply-add operation accumulating at twice the width of the multiplies. This transformation typically occurs after training, and it can reduce DNN accuracy by a few percentage points (Bhattacharya and Lane, 2016).

Summary of DNNs

Even this quick overview suggests that DSAs for DNNs will need to perform at least these matrix-oriented operations well: vector-matrix multiply, matrix-matrix multiply, and stencil computations. They will also need support for the nonlinear functions, which include at a minimum ReLU, Sigmoid, and tanh. These modest requirements still leave open a very large design space, which the next four sections explore.

7.4 Google's Tensor Processing Unit, an Inference Data Center Accelerator

The Tensor Processing Unit (*TPU*)[1] is Google's first custom ASIC DSA for WSCs. Its domain is the inference phase of DNNs, and it is programmed using the TensorFlow framework, which was designed for DNNs. The first TPU was been deployed in Google data centers in 2015.

The heart of the TPU is a 65,536 (256 × 256) 8-bit ALU Matrix Multiply Unit and a large software-managed on-chip memory. The TPU's single-threaded, deterministic execution model is a good match to the 99th-percentile response-time requirement of the typical DNN inference application.

TPU Origin

Starting as far back as 2006, Google engineers had discussions about deploying GPUs, FPGAs, or custom ASICs in their data centers. They concluded that the few applications that could run on special hardware could be done virtually for free using the excess capacity of the large data centers, and it's hard to improve on free. The conversation changed in 2013 when it was projected that if people used voice search for three minutes a day using speech recognition DNNs, it would have required Google's data centers to double in order to meet computation demands. That would be very expensive to satisfy with conventional CPUs. Google then started a high-priority project to quickly produce a custom ASIC for inference (and bought off-the-shelf GPUs for training). The goal was to improve cost-performance by 10 × over GPUs. Given this mandate, the TPU was designed, verified (Steinberg, 2015), built, and deployed in data centers in just 15 months.

TPU Architecture

To reduce the chances of delaying deployment, the TPU was designed to be a coprocessor on the PCIe I/O bus, which allows it to be plugged into existing servers. Moreover, to simplify hardware design and debugging, the host server sends instructions over the PCIe bus directly to the TPU for it to execute, rather than having the TPU fetch the instructions. Thus the TPU is closer in spirit to an FPU (floating-point unit) coprocessor than it is to a GPU, which fetches instructions from its memory.

Figure 7.12 shows the block diagram of the TPU. The host CPU sends TPU instructions over the PCIe bus into an instruction buffer. The internal blocks are typically connected together by 256-byte-wide (2048-bits) paths. Starting in the upper-right corner, the *Matrix Multiply Unit* is the heart of the TPU. It contains

---

[1] This section is based on the paper "In-Datacenter Performance Analysis of a Tensor Processing Unit" Jouppi et al., 2017, of which one of your book authors was a coauthor.

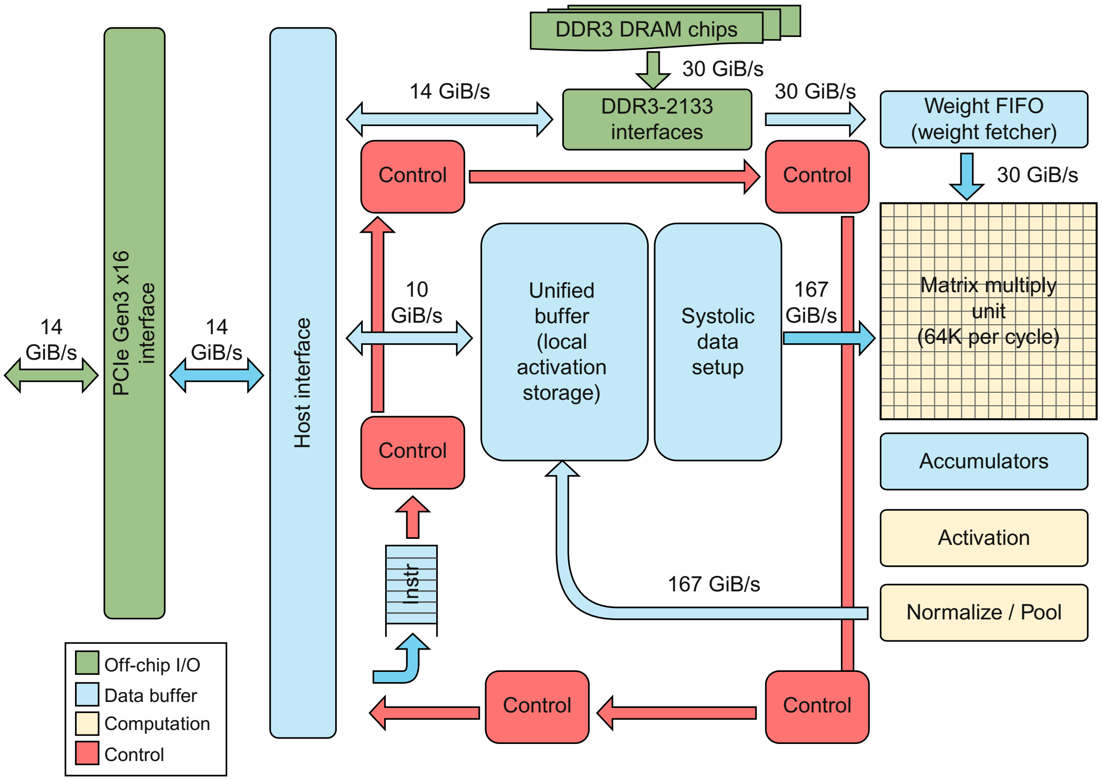

DDR3 DRAM chips
30 GiB/s
14 GiB/s
DDR3-2133 interfaces
30 GiB/s
Weight FIFO (weight fetcher)
Control
Control
30 GiB/s
PCIe Gen3 x16 interface
Host interface
14 GiB/s
14 GiB/s
10 GiB/s
Unified buffer (local activation storage)
Systolic data setup
167 GiB/s
Matrix multiply unit (64K per cycle)
Control
Accumulators
Activation
Instr
167 GiB/s
Normalize / Pool
Control
Control
Off-chip I/O
Data buffer
Computation
Control

Figure 7.12 TPU Block Diagram. The PCIe bus is Gen3 ×16. The main computation part is the light-shaded Matrix Multiply Unit in the upper-right corner. Its inputs are the medium-shaded Weight FIFO and the medium-shaded Unified Buffer and its output is the medium-shaded Accumulators. The light-shaded Activation Unit performs the nonlinear functions on the Accumulators, which go to the Unified Buffer.

256 × 256 ALUs that can perform 8-bit multiply-and-adds on signed or unsigned integers. The 16-bit products are collected in the 4 MiB of 32-bit *Accumulators* below the matrix unit. When using a mix of 8-bit weights and 16-bit activations (or vice versa), the Matrix Unit computes at half-speed, and it computes at a quarter-speed when both are 16 bits. It reads and writes 256 values per clock cycle and can perform either a matrix multiply or a convolution. The nonlinear functions are calculated by the *Activation* hardware.

The weights for the matrix unit are staged through an on-chip *Weight FIFO* that reads from an off-chip 8 GiB DRAM called *Weight Memory* (for inference, weights are read-only; 8 GiB supports many simultaneously active models). The intermediate results are held in the 24 MiB on-chip *Unified Buffer*, which can serve as inputs to the Matrix Multiply Unit. A programmable DMA controller transfers data to or from CPU Host memory and the Unified Buffer.

TPU Instruction Set Architecture

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The TPU does not have a program counter, and it has no branch instructions; instructions are sent from the host CPU. The clock cycles per instruction (CPI) of these CISC instructions are typically 10–20. It has about a dozen instructions overall, but these five are the key ones:

1. `Read_Host_Memory` reads data from the CPU host memory into the Unified Buffer.
2. `Read_Weights` reads weights from Weight Memory into the Weight FIFO as input to the Matrix Unit.
3. `MatrixMultiply/Convolve` causes the Matrix Multiply Unit to perform a matrix-matrix multiply, a vector-matrix multiply, an element-wise matrix multiply, an element-wise vector multiply, or a convolution from the Unified Buffer into the Accumulators. A matrix operation takes a variable-sized B*256 input, multiplies it by a 256×256 constant input, and produces a B*256 output, taking B pipelined cycles to complete. For example, if the input were 4 vectors of 256 elements, B would be 4, so it would take 4 clock cycles to complete.
4. `Activate` performs the nonlinear function of the artificial neuron, with options for ReLU, Sigmoid, tanh, and so on. Its inputs are the Accumulators, and its output is the Unified Buffer.
5. `Write_Host_Memory` writes data from the Unified Buffer into the CPU host memory.

The other instructions are alternate host memory read/write, set configuration, two versions of synchronization, interrupt host, debug-tag, nop, and halt. The CISC MatrixMultiply instruction is 12 bytes, of which 3 are Unified Buffer address; 2 are accumulator address; 4 are length (sometimes 2 dimensions for convolutions); and the rest are opcode and flags.

The goal is to run whole inference models in the TPU to reduce interactions with the host CPU and to be flexible enough to match the DNN needs of 2015 and beyond, instead of just what was required for 2013 DNNs.

TPU Microarchitecture

The microarchitecture philosophy of the TPU is to keep the Matrix Multiply Unit busy. The plan is to hide the execution of the other instructions by overlapping their execution with the `MatrixMultiply` instruction. Thus each of the preceding four general categories of instructions have separate execution hardware (with read and write host memory combined into the same unit). To increase instruction parallelism further, the `Read_Weights` instruction follows the decoupled access/

execute philosophy (Smith, 1982b) in that they can complete after sending their addresses but before the weights are fetched from Weight Memory. The matrix unit has not-ready signals from the Unified Buffer and the Weight FIFO that will cause the matrix unit to stall if their data are not yet available.

Note that a TPU instruction can execute for many clock cycles, unlike the traditional RISC pipeline with one clock cycle per stage.

Because reading a large SRAM is much more expensive than arithmetic, the Matrix Multiply Unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer (Kung and Leiserson, 1980; Ramacher et al., 1991; Ovtcharov et al., 2015b). A *systolic array* is a two-dimensional collection of arithmetic units that each independently compute a partial result as a function of inputs from other arithmetic units that are considered upstream to each unit. It relies on data from different directions arriving at cells in an array at regular intervals where they are combined. Because the data flows through the array as an advancing wave front, it is similar to blood being pumped through the human circulatory system by the heart, which is the origin of the systolic name.

Figure 7.13 demonstrates how a systolic array works. The six circles at the bottom are the multiply-accumulate units that are initialized with the weights *wi*. The staggered input data *xi* are shown coming into the array from above. The 10 steps of the figure represent 10 clock cycles moving down from top to bottom of the page. The systolic array passes the inputs down and the products and sums to the right. The desired sum of products emerges as the data completes its path through the systolic array. Note that in a systolic array, the input data is read only once from memory, and the output data is written only once to memory.

In the TPU, the systolic array is rotated. Figure 7.14 shows that the weights are loaded from the top and the input data flows into the array in from the left. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wave front. The weights are preloaded and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and after a feed delay, they update one location of each of 256 accumulator memories. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.

TPU Implementation

The TPU chip was fabricated using the 28-nm process. The clock rate is 700 MHz. Figure 7.15 shows the floor plan of the TPU. Although the exact die size is not revealed, it is less than half the size of an Intel Haswell server microprocessor, which is 662 mm^2.

The 24 MiB Unified Buffer is almost a third of the die, and the Matrix Multiply Unit is a quarter, so the datapath is nearly two-thirds of the die. The 24 MiB size was picked in part to match the pitch of the Matrix Unit on the die and, given the

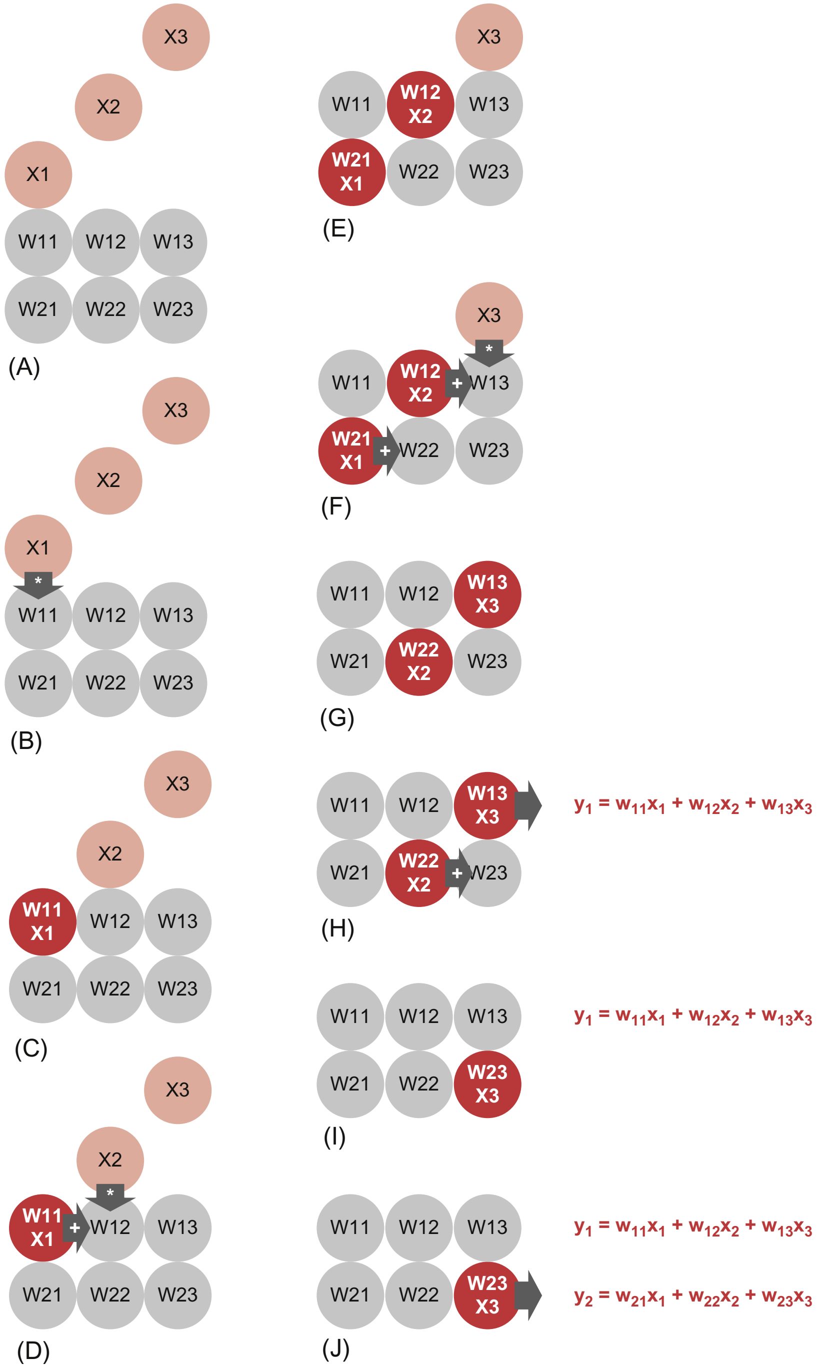

X3
X2
X1
W11 W12 W13
W21 W22 W23
(A)
X3
X2
X1
*
W11 W12 W13
W21 W22 W23
(B)
X3
X2
W11 X1
W12 W13
W21 W22 W23
(C)
X3
X2
*
W11 X1
+
W12 W13
W21 W22 W23
(D)
X3
W11
W12 X2
W13
W21 X1
W22 W23
(E)
X3
*
W11
W12 X2
+
W13
W21 X1
+
W22 W23
(F)
W11 W12
W13 X3
W21
W22 X2
W23
(G)
W11 W12
W13 X3
W21
W22 X2
+
W23
(H)
$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3$
W11 W12 W13
W21 W22
W23 X3
(I)
$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3$
W11 W12 W13
W21 W22
W23 X3
(J)
$y_1 = w_{11}x_1 + w_{12}x_2 + w_{13}x_3$
$y_2 = w_{21}x_1 + w_{22}x_2 + w_{23}x_3$

Figure 7.13 Example of systolic array in action, from top to bottom on the page. In this example, the six weights are already inside the multiply-accumulate units, as is the norm for the TPU. The three inputs are staggered in time to get the desired effect, and in this example are shown coming in from the top. (In the TPU, the data actually comes in from the left.) The array passes the data down to the next element and the result of the computation to the right to the next element. At the end of the process, the sum of products is found to the right. Drawings courtesy of Yaz Sato.

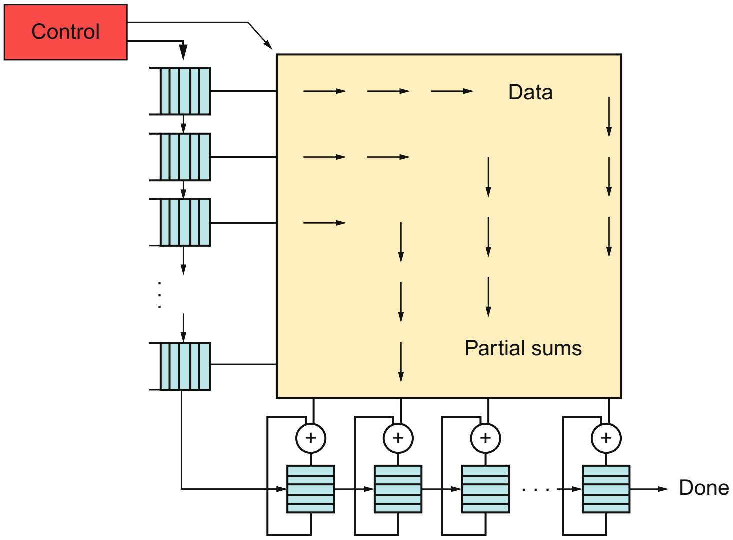

Control
Data
Partial sums
+
Done

Figure 7.14 Systolic data flow of the Matrix Multiply Unit.

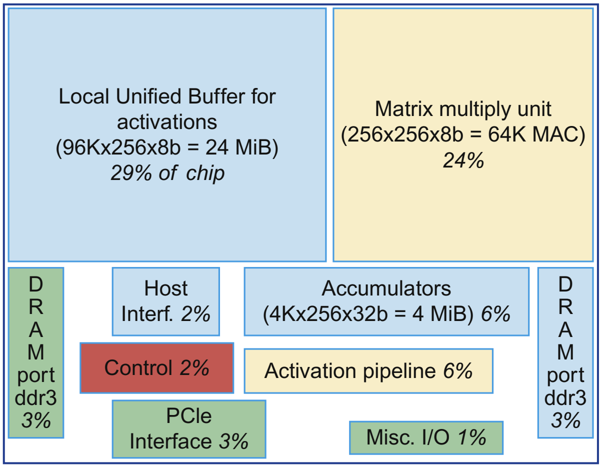

Local Unified Buffer for activations (96Kx256x8b = 24 MiB) *29% of chip*
Matrix multiply unit (256x256x8b = 64K MAC) *24%*
DRAM port ddr3 *3%*
Host Interf. *2%*
Accumulators (4Kx256x32b = 4 MiB) *6%*
DRAM port ddr3 *3%*
Control *2%*
Activation pipeline *6%*
PCIe Interface *3%*
Misc. I/O *1%*

Figure 7.15 Floor plan of TPU die. The shading follows Figure 7.14. The light data buffers are 37%, the light computation units are 30%, the medium I/O is 10%, and the dark control is just 2% of the die. Control is much larger (and much more difficult to design) in a CPU or GPU. The unused white space is a consequence of the emphasis on time to tape-out for the TPU.

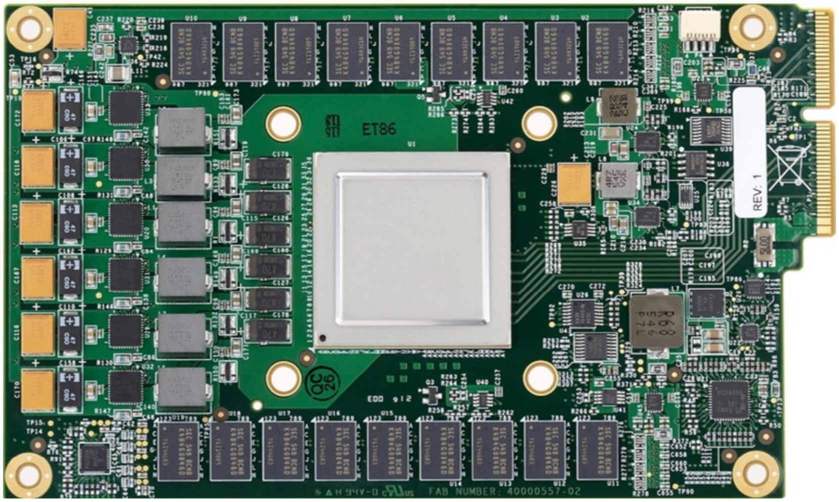


Figure 7.16 TPU printed circuit board. It can be inserted into the slot for an SATA disk in a server, but the card uses the PCIe bus.

short development schedule, in part to simplify the compiler. Control is just 2%. Figure 7.16 shows the TPU on its printed circuit card, which inserts into existing servers in a SATA disk slot.

TPU Software

The TPU software stack had to be compatible with that developed for CPUs and GPUs so that applications could be ported quickly. The portion of the application run on the TPU is typically written using TensorFlow and is compiled into an API that can run on GPUs or TPUs (Larabel, 2016). Figure 7.17 shows TensorFlow code for a portion of an MLP.

Like GPUs, the TPU stack is split into a User Space Driver and a Kernel Driver. The Kernel Driver is lightweight and handles only memory management and interrupts. It is designed for long-term stability. The User Space driver changes frequently. It sets up and controls TPU execution, reformats data into TPU order, and translates API calls into TPU instructions and turns them into an application binary. The User Space driver compiles a model the first time it is evaluated, caching the program image and writing the weight image into the TPU Weight Memory; the second and following evaluations run at full speed. The TPU runs most models completely from inputs to outputs, maximizing the ratio of TPU compute time to I/O time. Computation is often done one layer at a time, with overlapped execution allowing the matrix unit to hide most noncritical path operations.

```
# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with ReLU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with ReLU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

Figure 7.17 Portion of the TensorFlow program for the MNIST MLP. It has two hidden 256 × 256 layers, with each layer using a ReLU as its nonlinear function.

Improving the TPU

The TPU architects looked at variations of the microarchitecture to see whether they could have improved the TPU.

Like an FPU, the TPU coprocessor has a relatively easy microarchitecture to evaluate, so the TPU architects created a performance model and estimated performance as the memory bandwidth, the matrix unit size, and the clock rate and number of accumulators varied. Measurements using TPU hardware counters found that the modeled performance was on average within 8% of the hardware.

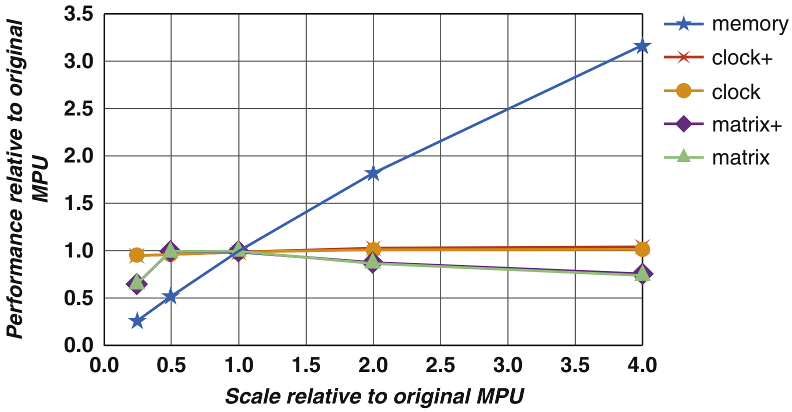


Figure 7.18 Performance as metrics scale from 0.25× to 4×: memory bandwidth, clock rate+accumulators, clock rate, matrix unit dimension+accumulators, and one dimension of the square matrix unit This is the average performance calculated from six DNN applications in Section 7.9. The CNNs tend to be computation-bound, but the MLPs and LSTMs are memory-bound. Most applications benefit from a faster memory, but a faster clock makes little difference, and a bigger matrix unit actually hurts performance. This performance model is only for code running inside the TPU and does not factor in the CPU host overhead.

Figure 7.18 shows the performance sensitivity of the TPU as these parameters scale over the range for 0.25 × to 4 ×. (Section 7.9 lists the benchmarks used.) In addition to evaluating the impact of only raising clock rates (*clock* in Figure 7.18), Figure 7.18 also plots a design (*clock*+) that increases the clock rate and scales the number of accumulators correspondingly so that the compiler can keep more memory references in flight. Likewise, Figure 7.18 plots matrix unit expansion if the number of accumulators increase with the square of the rise in one dimension (*matrix*+), because the matrix grows in both dimensions, as well as only increasing the matrix unit (*matrix*).

First, increasing memory bandwidth (*memory*) has the biggest impact: performance improves 3 × on average when memory bandwidth increases 4 ×, because it reduces the time waiting for weight memory. Second, clock rate has little benefit on average with or without more accumulators. Third, the average performance in Figure 7.18 slightly *degrades* when the matrix unit expands from 256 × 256 to 512 × 512 for all applications, whether or not they get more accumulators. The issue is analogous to internal fragmentation of large pages, only worse because it's in two dimensions.

Consider the 600 × 600 matrix used in LSTM1. With a 256 × 256 matrix unit, it takes nine steps to tile 600 × 600, for a total of 18 μs of time. The larger 512 × 512 unit requires only four steps, but each step takes four times longer, or 32 μs of time. The TPU's CISC instructions are long, so decode is insignificant and does not hide the overhead of loading from the DRAM.

Given these insights from the performance model, the TPU architects next evaluated an alternative and hypothetical TPU that they might have designed in the same process technology if they'd had more than 15 months to do so. More aggressive logic synthesis and block design might have increased the clock rate by 50%. The architects found that designing an interface circuit for GDDR5 memory, as used by the K80, would improve Weight Memory bandwidth by more than a factor of five. As Figure 7.18 shows, increasing clock rate to 1050 MHz, but not helping memory, made almost no change in performance. If the clock is left at 700 MHz, but it uses GDDR5 instead for Weight Memory, performance is increased by 3.2 ×, even accounting for the host CPU overhead of invoking the DNN on the revised TPU. Doing both does not improve average performance further.

Summary: How TPU Follows the Guidelines

Despite living on an I/O bus and having relatively little memory bandwidth that limits full utilization of the TPU, a small fraction of a big number can, nonetheless, be relatively large. As we will see in Section 7.9, the TPU delivered on its goal of a tenfold improvement in cost-performance over the GPU when running DNN inference applications. Moreover, a redesigned TPU with the only change being a switch to the same memory technology as in the GPU would be three times faster.

One way to explain the TPU's success is to see how it followed the guidelines in Section 7.2.

1. *Use dedicated memories to minimize the distance over which data is moved.*
   The TPU has the 24 MiB Unified Buffer that holds the intermediate matrices and vectors of MLPs and LSTMs and the feature maps of CNNs. It is optimized for accesses of 256 bytes at a time. It also has the 4 MiB Accumulators, each 32-bits wide, that collect the output of the Matrix Unit and act as input to the hardware that calculates the nonlinear functions. The 8-bit weights are stored in a separate off-chip weight memory DRAM and are accessed via an on-chip weight FIFO. In contrast, all these types and sizes of data would exist in redundant copies at several levels of the inclusive memory hierarchy of a general-purpose CPU.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   The TPU offers 28 MiB of dedicated memory and 65,536 8-bit ALUs, which means it has about 60% of the memory and 250 times as many ALUs as a server-class CPU, despite being half its size and power (see Section 7.9). Compared to a server-class GPU, the TPU has 3.5 times the on-chip memory and 25 times as many ALUs.
3. *Use the easiest form of parallelism that matches the domain.*
   The TPU delivers its performance via a two-dimensional SIMD parallelism with its 256 × 256 Matrix Multiply Unit, which is internally pipelined with a systolic organization, plus a simple overlapped execution pipeline of its

instructions. GPUs rely instead on multiprocessing, multithreading, and one-dimensional SIMD, and CPUs rely on multiprocessing, out-of-order execution, and one-dimensional SIMD.

4. *Reduce data size and type to the simplest needed for the domain.*
   The TPU computes primarily on 8-bit integers, although it supports 16-bit integers and accumulates in 32-bit integers. CPUs and GPUs also support 64-bit integers and 32-bit and 64-bit floating point.
5. *Use a domain-specific programming language to port code to the DSA.*
   The TPU is programmed using the TensorFlow programming framework, whereas GPUs rely on CUDA and OpenCL and CPUs must run virtually everything.

7.5 Microsoft Catapult, a Flexible Data Center Accelerator

At the same time that Google was thinking about deploying a custom ASIC in its data centers, Microsoft was considering accelerators for theirs. The Microsoft perspective was that any solution had to follow these guidelines:

- It had to preserve homogeneity of servers to enable rapid redeployment of machines and to avoid making maintenance and scheduling even more complicated, even if that notion is a bit at odds with the concept of DSAs.
- It had to scale to applications that might need more resources than could fit into a single accelerator without burdening all applications with multiple accelerators.
- It needed to be power-efficient.
- It couldn't become a dependability problem by being a single point of failure.
- It had to fit within the available spare space and power in existing servers.
- It could not hurt data center network performance or reliability.
- The accelerator had to improve the cost-performance of the server.

The first rule prevented deploying an ASIC that helped only some applications on some servers, which was the decision that Google made.

Microsoft started a project called Catapult that placed an FPGA on a PCIe bus board into data center servers. These boards have a dedicated network for applications that need more than one FPGA. The plan was to use the flexibility of the FPGA to tailor its use for varying applications both on different servers and to reprogram the same server to accelerate distinct applications over time. This plan increased the return on its investment of the accelerator. Another advantage of FPGAs is that they should have lower NRE than ASICs, which could again improve return on investment. We discuss two generations of Catapult, showing how the design evolved to meet the needs of WSCs.

One interesting upside of FPGAs is that each application—or even each phase of an application—can be thought of as its own DSA, so in this section, we get to see many examples of novel architectures in one hardware platform.

Catapult Implementation and Architecture

Figure 7.19 shows a PCIe board that Microsoft designed to fit within its servers, which limited power and cooling to 25 W. This constraint led to the selection of the 28-nm Altera Stratix V D5 FPGA for its first implementation of Catapult. The board also has 32 MiB of flash memory and includes two banks of DDR3-1600 DRAM with a total capacity of 8 GiB. The FPGA has 3926 18-bit ALUs, 5 MiB of on-chip memory, and 11 GB/s bandwidth to DDR3 DRAMs.

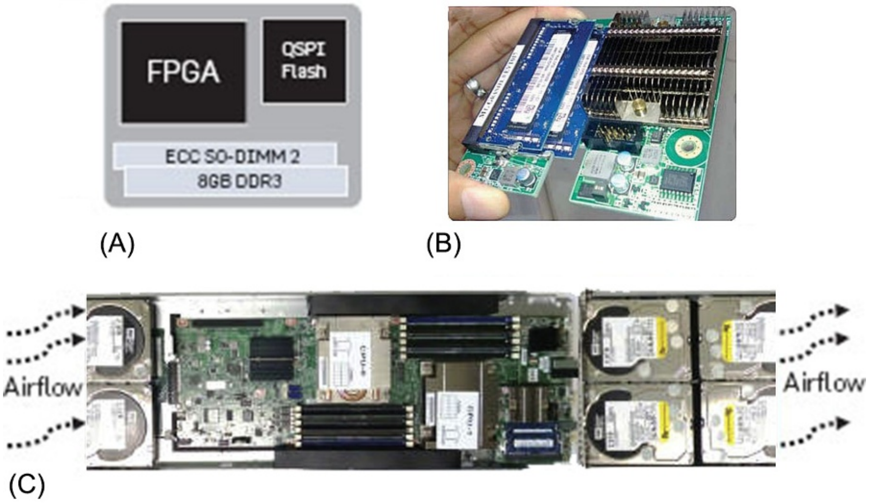


Figure 7.19 The Catapult board design. (A) shows the block diagram, and (B) is a photograph of both sides of the board, which is 10 cm × 9 cm × 16 mm. The PCIe and inter-FPGA network are wired to a connector on the bottom of the board that plugs directly into the motherboard. (C) is a photograph of the server, which is 1U (4.45 cm) high and half a standard rack wide. Each server has two 12-core Intel Sandy Bridge Xeon CPUs, 64 GiB of DRAM, 2 solid-state drives, 4 hard-disk drives, and a 10-Gbit Ethernet network card. The highlighted rectangle on the right in (C) shows the location of the Catapult FPGA board on the server. The cool air is sucked in from the left in (C), and the hot air exhausts to the right, which passes over the Catapult board. This hot spot and the amount of the power that the connector could deliver mean that the Catapult board is limited to 25 watts. Forty-eight servers share an Ethernet switch that connects to the data center network, and they occupy half of a data center rack.

Each of the 48 servers in half of a data center rack contains a Catapult board. Catapult follows the preceding guidelines about supporting applications that need more than a single FPGA without affecting the performance of the data center network. It adds a separate low-latency 20 Gbit/s network that connects 48 FPGAs. The network topology is a two-dimensional 6 × 8 torus network.

To follow the guideline about not being a single point of failure, this network can be reconfigured to operate even if one of the FPGAs fails. The board also has SECDED protection on all memories outside the FPGA, which is required for large-scale deployment in a data center.

Because FPGAs use a great deal of memory on the chip to deliver programmability, they are more vulnerable than ASICs to *single-event upsets (SEUs)* because of radiation as the process geometries shrink. The Altera FPGA in Catapult boards includes mechanisms to detect and correct SEUs inside the FPGA and reduces the chances of SEUs by periodically scrubbing the FPGA configuration state.

The separate network has an added benefit of reducing the variability of communication performance as compared to a data center network. Network unpredictability increases tail latency—which is especially detrimental for applications that face end users—so a separate network makes it easier to successfully offload work from the CPU to the accelerator. This FPGA network can run a much simpler protocol than in the data center because the error rates are considerably lower and the network topology is well defined.

Note that resiliency requires care when reconfiguring FPGAs so that they neither appear as failed nodes nor crash the host server or corrupt their neighbors. Microsoft developed a high-level protocol for ensuring safety when reconfiguring one or more FPGAs.

Catapult Software

Possibly the biggest difference between Catapult and the TPU is having to program in a hardware-description language such as Verilog or VHDL. As the Catapult authors write (Putnam et al., 2016):

> Going forward, the biggest obstacle to widespread adoption of FPGAs in the datacenter is likely to be programmability. FPGA development still requires extensive hand-coding in Register Transfer Level and manual tuning.

To reduce the burden of programming Catapult FPGAs, the Register Transfer Level (RTL) code is divided into the *shell* and the *role*, as Figure 7.20 shows. The shell code is like the system library on an embedded CPU. It contains the RTL code that will be reused across applications on the same FPGA board, such as data marshaling, CPU-to-FPGA communication, FPGA-to-FPGA communication, data movement, reconfiguration, and health monitoring. The shell RTL code is 23% of the Altera FPGA. The role code is the application logic, which the Catapult programmer writes using the remaining 77% of the FPGA resources. Having a shell has the added benefit of offering a standard API and standard behavior across applications.

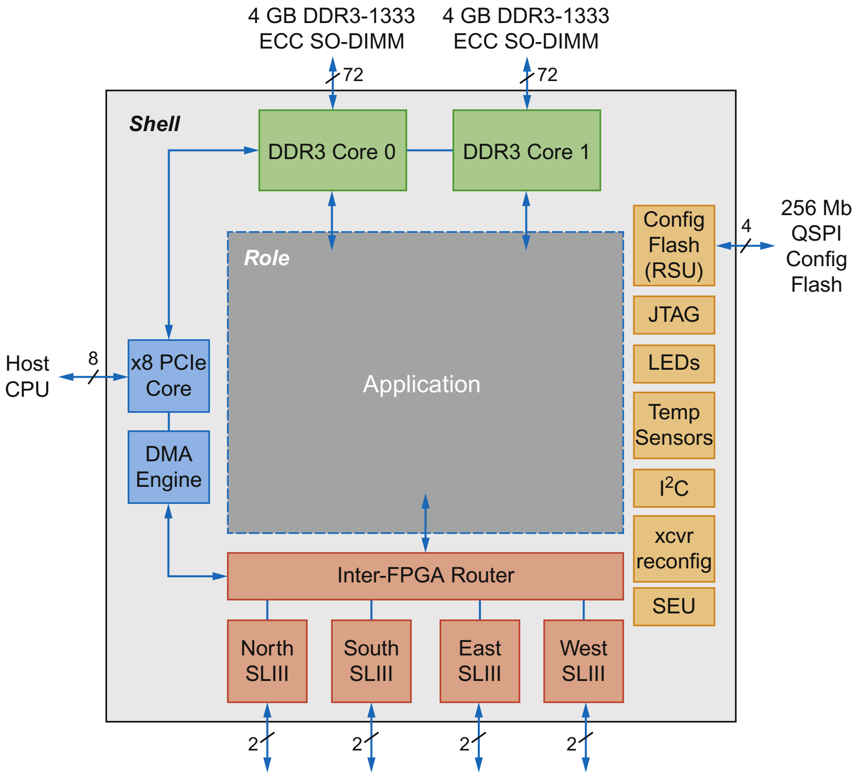

4 GB DDR3-1333
ECC SO-DIMM
4 GB DDR3-1333
ECC SO-DIMM
72
72
Shell
DDR3 Core 0
DDR3 Core 1
Config
Flash
(RSU)
4
256 Mb
QSPI
Config
Flash
Role
JTAG
Host
CPU
8
x8 PCIe
Core
Application
LEDs
Temp
Sensors
DMA
Engine
I²C
xcvr
reconfig
Inter-FPGA Router
SEU
North
SLIII
South
SLIII
East
SLIII
West
SLIII
2
2
2
2

Figure 7.20 Components of Catapult shell and role split of the RTL code.

CNNs on Catapult

Microsoft developed a configurable CNN accelerator as an application for Catapult. Configuration parameters include the number of neural network layers, the dimension of those layers, and even the numerical precision to be used. Figure 7.21 shows the block diagram of the CNN accelerator. Its key features are:

- Run-time configurable design, without requiring recompilation using the FPGA tools.
- To minimize memory accesses, it offers efficient buffering of CNN data structures (see Figure 7.21).
- A two-dimensional array of Processing Elements (PEs) that can scale up to thousands of units.

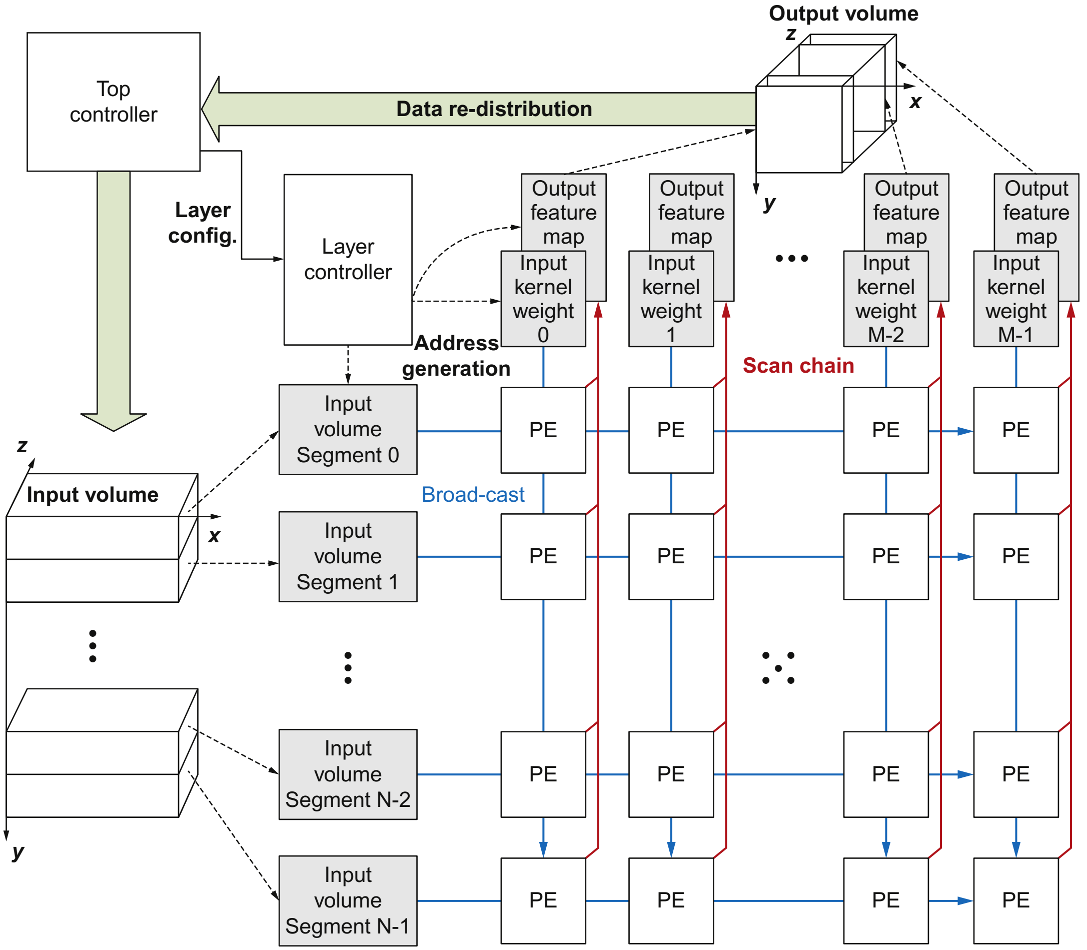

Output volume
z
x
y
Top controller
Data re-distribution
Layer config.
Layer controller
Address generation
Output feature map
Input kernel weight 0
Input kernel weight 1
Input kernel weight M-2
Input kernel weight M-1
Scan chain
Input volume
Input volume Segment 0
Input volume Segment 1
Input volume Segment N-2
Input volume Segment N-1
Broad-cast
PE

Figure 7.21 CNN Accelerator for Catapult. The Input Volume of the left correspond to Layer[*i*−1] on the left of Figure 7.20, with NumFM[*i*−1] corresponding to y and DimFM[*i*−1] corresponding to *z*. Output Volume at the top maps to Layer[*i*], with z mapping to NumFM[*i*] and DimFM[*i*] mapping to *x*. The next figure shows the inside of the Processing Element (PE).

Images are sent to DRAM and then input into a multibank buffer in the FPGA. The inputs are sent to multiple PEs to perform the stencil computations that produce the output feature maps. A controller (upper left in Figure 7.21) orchestrates the flow of data to each PE. The final results are then recirculated to the input buffers to compute the next layer of the CNN.

Like the TPU, the PEs are designed to be used as a systolic array. Figure 7.22 shows the details of the PE design.

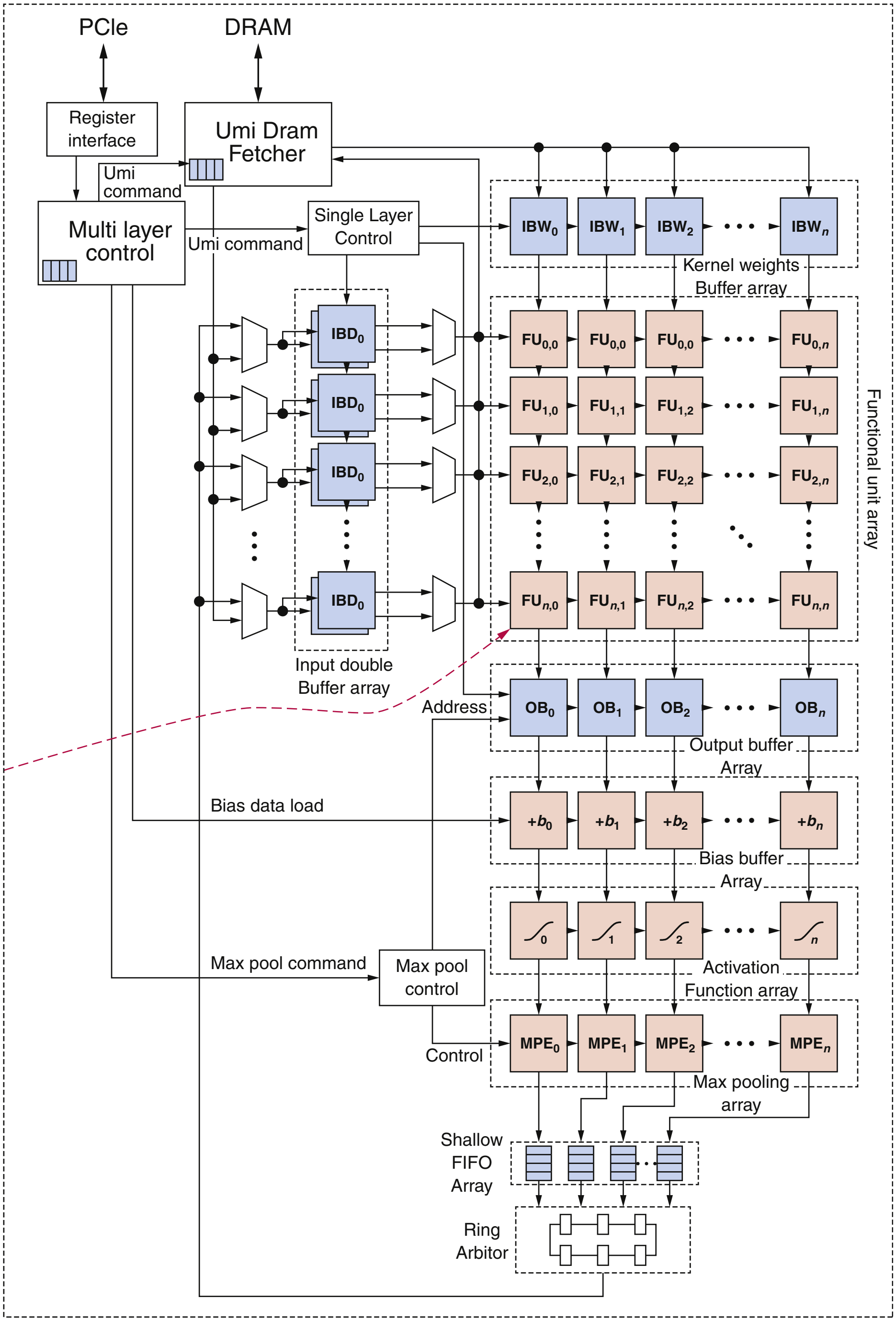

PCIe
DRAM
Register interface
Umi Dram Fetcher
Umi command
Multi layer control
Single Layer Control
Umi command
IBW$_0$
IBW$_1$
IBW$_2$
IBW$_n$
Kernel weights Buffer array
IBD$_0$
Input double Buffer array
FU$_{0,0}$
FU$_{0,n}$
FU$_{1,0}$
FU$_{1,1}$
FU$_{1,2}$
FU$_{1,n}$
FU$_{2,0}$
FU$_{2,1}$
FU$_{2,2}$
FU$_{2,n}$
FU$_{n,0}$
FU$_{n,1}$
FU$_{n,2}$
FU$_{n,n}$
Functional unit array
Address
OB$_0$
OB$_1$
OB$_2$
OB$_n$
Output buffer Array
Bias data load
$+b_0$
$+b_1$
$+b_2$
$+b_n$
Bias buffer Array
Activation Function array
Max pool command
Max pool control
Control
MPE$_0$
MPE$_1$
MPE$_2$
MPE$_n$
Max pooling array
Shallow FIFO Array
Ring Arbitor

Figure 7.22 The Processing Element (PE) of the CNN Accelerator for Catapult in Figure 7.21. The two-dimension Functional Units (FU) consist of just an ALU and a few registers.

Search Acceleration on Catapult

The primary application to test the return on investment of Catapult was a critical function of the Microsoft Bing search engine called *ranking*. It ranks the order of the results from a search. The output is a document score, which determines the position of the document on the web page that is presented to the user. The algorithm has three stages:

1. *Feature Extraction* extracts thousands of interesting features from a document based on the search query, such as the frequency that the query phrase appears in a document.
2. *Free-Form Expressions* calculates thousands of combinations of features from the prior stage.
3. *Machine-Learned Scoring* uses machine-learning algorithms to evaluate the features from the first two stages to calculate a floating-point score of a document that is returned to the host search software.

The Catapult implementation of ranking produces identical results to equivalent Bing software, even reproducing known bugs!

Taking advantage of one of the preceding guidelines, the ranking function does not have to fit within a single FPGA. Here is how the ranking stages are split across eight FPGAs:

- One FPGA does Feature Extraction.
- Two FPGAs do Free-Form Expressions.
- One FPGA does a compression stage that increases scoring engine efficiency.
- Three FPGA do Machine-Learned Scoring.

The remaining FPGA is a spare used to tolerate faults. Using multiple FPGAs for one application works well because of the dedicated FPGA network.

Figure 7.23 shows the Feature Extraction stage organization. It uses 43 feature-extraction state machines to compute in parallel 4500 features per document-query pair.

Next is the following Free-Form Expressions stage. Rather than implement the functions directly in gates or in state machines, Microsoft developed a 60-core processor that overcomes long-latency operations with multithreading. Unlike a GPU, Microsoft's processor does not require SIMD execution. It has three features that let it match the latency target:

1. Each core supports four simultaneous threads where one can stall on a long operation but the others can continue. All functional units are pipelined, so they can accept a new operation every clock cycle.
2. Threads are statically prioritized using a priority encoder. Expressions with the longest latency use thread slot 0 on all cores, then the next slowest is in slot 1 on all cores, and so on.

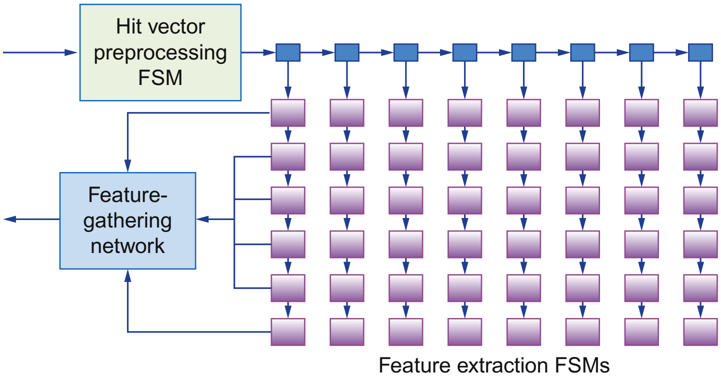


Figure 7.23 The architecture of FPGA implementation of the Feature Extraction stage. A hit vector, which describes the locations of query words in each document, is streamed into the hit vector preprocessing state machine and then split into control and data tokens. These tokens are issued in parallel to the 43 unique feature state machines. The feature-gathering network collects generated feature and value pairs and forwards them to the following Free-Form Expressions stage.

3. Expressions that are too large to fit in the time allocated for a single FPGA can be split across the two FPGAs used for free-form expressions.

One cost of the reprogrammability in an FPGA is a slower clock rate than custom chips. Machine-Learned Scoring uses two forms of parallelism to try to overcome that disadvantage. The first is to have a pipeline that matches the available pipeline parallelism in the application. For ranking, the limit is 8 μs per stage. The second version of parallelism is the rarely seen *multiple instruction streams, single data stream (MISD)* parallelism, where a large number of independent instruction streams operate in parallel on a single document.

Figure 7.24 shows the performance of the ranking function on Catapult. As we will see in Section 7.9, user-facing applications often have rigid response times; it doesn't matter how high the throughput is if the application misses the deadline. The *x*-axis shows the response-time limit, with 1.0 as the cutoff. At this maximum latency, Catapult is 1.95 times as fast as the host Intel server.

Catapult Version 1 Deployment

Before populating a whole warehouse-scale computer with tens of thousands of servers, Microsoft did a test deployment of 17 full racks, which contained 17 × 48 × 2 or 1632 Intel servers. The Catapult cards and network links were tested at manufacture and system integration, but at deployment, seven of the 1632 cards failed (0.43%), and one of the 3264 FPGA network links (0.03%) was defective. After several months of deployment, nothing else failed.

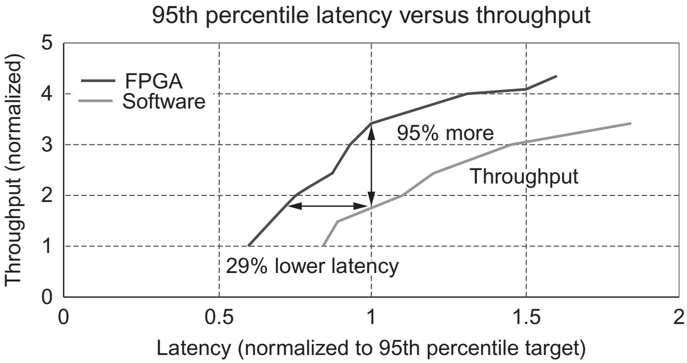


Figure 7.24 Performance for the ranking function on Catapult for a given latency bound. The *x*-axis shows the response time for the Bing ranking function. The maximum response time at the 95th percentile for the Bing application on the *x*-axis is 1.0, so data points to the right may have a higher throughput but arrive too late to be useful. The *y*-axis shows the 95% throughputs on Catapult and pure software for a given response time. At a normalized response time of 1.0, Catapult has 1.95 the throughput of Intel server running in pure software mode. Stated alternatively, if Catapult matches the throughput that the Intel server has at 1.0 normalized response time, Catapult's response time is 29% less.

Catapult Version 2

Although the test deployment was successful, Microsoft changed the architecture for the real deployment to enable both Bing and Azure Networking to use the same boards and architecture (Caulfield et al., 2016). The main problem with the V1 architecture was that the independent FPGA network did not enable the FPGA to see and process standard Ethernet/IP packets, which prevented it from being used to accelerate the data center network infrastructure. In addition, the cabling was expensive and complicated, it was limited to 48 FPGAs, and the rerouting of traffic during certain failure patterns reduced performance and could isolate nodes.

The solution was to place the FPGA logically between the CPU and NIC, so that all network traffic goes through the FPGA. This "bump-on-a-wire" placement removes many weaknesses of the FPGA network in Catapult V1. Moreover, it enables the FPGAs to run their own low-latency network protocol that allows them to be treated as a global pool of all the FPGAs in the data center and even across data centers.

Three changes occurred between V1 and V2 to overcome the original concerns of Catapult applications interfering with data center network traffic. First, the data center network was upgraded from 10 Gbit/s to 40 Gbit/s, increasing the headroom. Second, Catapult V2 added a rate limiter for FPGA logic, ensuring that an FPGA application could not overwhelm the network. The final and perhaps most important change was that the networking engineers would now had their own use cases for the FPGA, given its bump-in-the-wire placement. That placement transformed these former interested bystanders into enthusiastic collaborators.

By deploying Catapult V2 in the majority of its new servers, Microsoft essentially has a second supercomputer composed of distributed FPGAs that shares the same network wires as the CPU servers and is at the same scale, as there is one FPGA per server. Figures 7.25 and 7.26 show the block diagram and the board for Catapult V2.

Catapult V2 follows the same shell and role split of the RTL to simplify programming, but at the time of publication, the shell uses almost half of the FPGA resources (44%) because of the more complicated network protocol that shares the data center network wires.

Catapult V2 is used for both Ranking acceleration and function network acceleration. In Ranking acceleration, rather than perform nearly all of the ranking function inside the FPGA, Microsoft implemented only the most compute-intensive portions and left the rest to the host CPU:

- The *feature functional unit (FFU)* is a collection of finite state machines that measure standard features in search, such as counting the frequency of a particular search term. It is similar in concept to the Feature Extraction stage of Catapult V1.

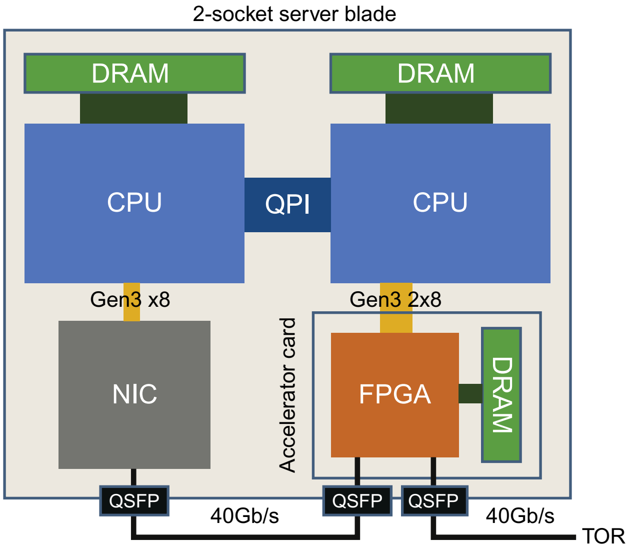


Figure 7.25 The Catapult V2 block diagram. All network traffic is routed through the FPGA to the NIC. There is also a PCIe connector to the CPUs, which allows the FPGA to be used as a local compute accelerator, as in Catapult V1.

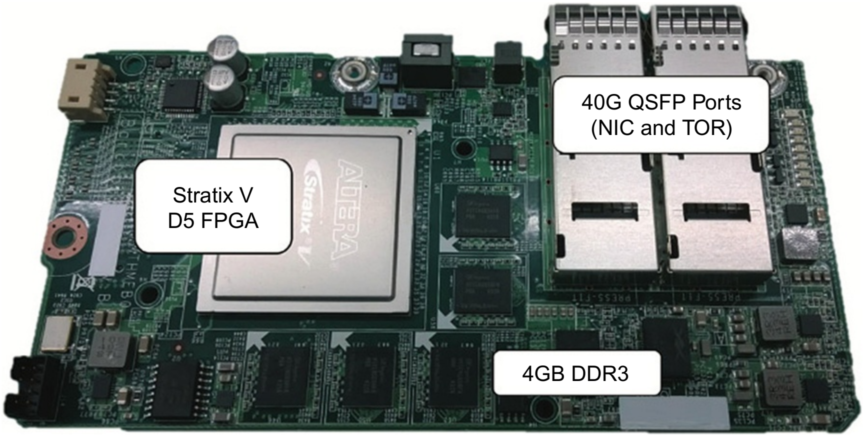


Figure 7.26 The Catapult V2 board uses a PCIe slot. It uses the same FPGA as Catapult V1 and has a TDP of 32 W. A 256-MB Flash chip holds the *golden image* for the FPGA that is loaded at power on, as well as one application image.

- The *dynamic programming feature (DPF)* creates a Microsoft proprietary set of features using dynamic programming and bears some similarity to the Free-Form Expressions stage of Catapult V1.

Both are designed so that they can use non-local FPGAs for these tasks, which simplifies scheduling.

Figure 7.27 shows the performance of Catapult V2 compared to software in a format similar to Figure 7.24. The throughput can now be increased 2.25 × without endangering latency, whereas the speedup was previously 1.95 ×. When ranking was deployed and measured in production, Catapult V2 had better tail latencies than software; that is, the FPGA latencies never exceeded the software latencies at any given demand despite being able to absorb twice the workload.

Summary: How Catapult Follows the Guidelines

Microsoft reported that adding Catapult V1 to the servers in the pilot phase increased the total cost of ownership (TCO) by less than 30%. Thus, for this application, the net gain in cost-performance for Ranking was at least 1.95/1.30, or a return on investment of about 1.5. Although no comment was made about TCO concerning Catapult V2, the board has a similar number of the same type of chips, so we might guess that the TCO is no higher. If so, the cost-performance of Catapult V2 is about 2.25/1.30, or 1.75 for Ranking.

Here is how Catapult followed the guidelines from Section 7.2.

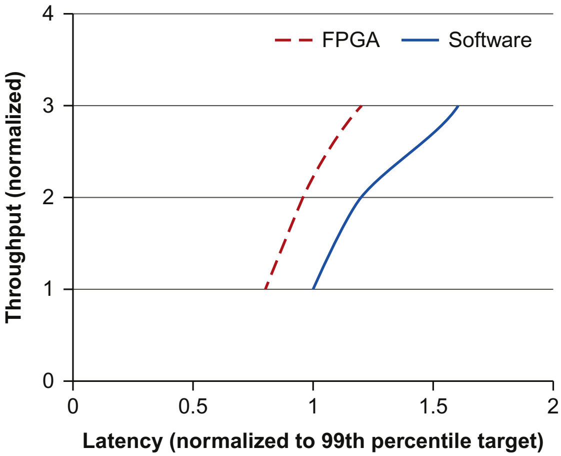


Figure 7.27 Performance for the ranking function on Catapult V2 in the same format as Figure 7.24. Note that this version measures 99th percentile while the earlier figure plots 95th percentile.

1. *Use dedicated memories to minimize the distance over which data is moved.* The Altera V FPGA has 5 MiB of memory on-chip, which an application can customize for its use. For example, for CNNs, it is used for the input and output feature maps of Figure 7.21.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.* The Altera V FPGA also has 3926 18-bit ALUs that are tailored to the application. For CNNs, they are used to create the systolic array that drives the Processing Elements in Figure 7.22, and they form the datapaths of the 60-core multiprocessor used by Free Form Expression stage of ranking.
3. *Use the easiest form of parallelism that matches the domain.* Catapult picks the form of parallelism that matches the application. For example, Catapult uses two-dimensional SIMD parallelism for the CNN application and MISD parallelism in the Machine Scoring phase stream Ranking.
4. *Reduce data size and type to the simplest needed for the domain.* Catapult can use whatever size and type of data that the application wants, from an 8-bit integer to a 64-bit floating point.
5. *Use a domain-specific programming language to port code to the DSA.* In this case, programming is done in the hardware register-transfer language (RTL) Verilog, which is an even less productive language than C or C++. Microsoft did not (and possibly could not) follow this guideline given its use of FPGAs.

Although this guideline concerns the one-time porting of an application from software to FPGA, applications are not frozen in time. Almost by definition, software evolves to add features or fix bugs, especially for something as important as web search.

Maintenance of successful programs can be most of software's development costs. Moreover, when programming in an RTL, software maintenance is even more burdensome. The Microsoft developers, like all others who use FPGAs as accelerators, hope that future advances in domain-specific languages and systems for hardware-software co-design will reduce the difficulty of programming FPGAs.

7.6 Intel Crest, a Data Center Accelerator for Training

The quotation by the Intel CEO that opens Section 7.3 came from the press release announcing that Intel was going to start shipping DSAs ("accelerants") for DNN. The first example was Crest, which was announced while we were writing this edition. Despite the limited details, we include it here because of the significance of a traditional microprocessor manufacturer like Intel taking this bold step of embracing DSAs.

Crest is aimed at DNN training. The Intel CEO said the goal is to accelerate DNN training a hundredfold over the next three years. Figure 7.6 shows that training can take a month. There is likely to be a demand to decrease the DNN training to just eight hours, which would be 100 times quicker than the CEO predicted. DNNs will surely become even more complex over the next 3 years, which will require a much greater training effort. Thus there seems little danger that a 100 × improvement in training is overkill.

Crest instructions operate on blocks of 32 × 32 matrices. Crest uses a number format called *flex point*, which is a scaled fixed-point representation: 32 × 32 matrices of 16-bit data share a single 5-bit exponent that is provided as part of the instruction set.

Figure 7.28 shows the block diagram of the Lake Crest chip. To compute on these matrices, Crest uses the12 processing clusters of Figure 7.28. Each cluster includes a large SRAM, a big linear algebra processing unit, and a small amount of logic for on- and off-chip routing. The four 8 GiB HBM2 DRAM modules offer 1 TB/s of memory bandwidth, which should lead to an attractive Roofline model for the Crest chip. In addition to high-bandwidth paths to main memory, Lake Crest supports high bandwidth interconnects directly between compute cores inside the processing clusters, which facilitates quick core-to-core communication without passing through shared memory. Lake Crest's goal is a factor of 10 improvement in training over GPUs.

Figure 7.28 shows 12 Inter-Chip Links (ICLs) and 2 Inter-Chip Controllers (ICCs), so Crest is clearly designed to allow many Crest chips to collaborate, similar in spirit to the dedicated network connecting the 48 FPGAs in Catapult. It's likely that the 100 × improvement in training will require ganging together several Crest chips.

7.7 Pixel Visual Core, a Personal Mobile Device Image Processing Unit

Pixel Visual Core is a programmable, scalable DSA intended for image processing and computer vision from Google, initially for cell phones and tablets running the

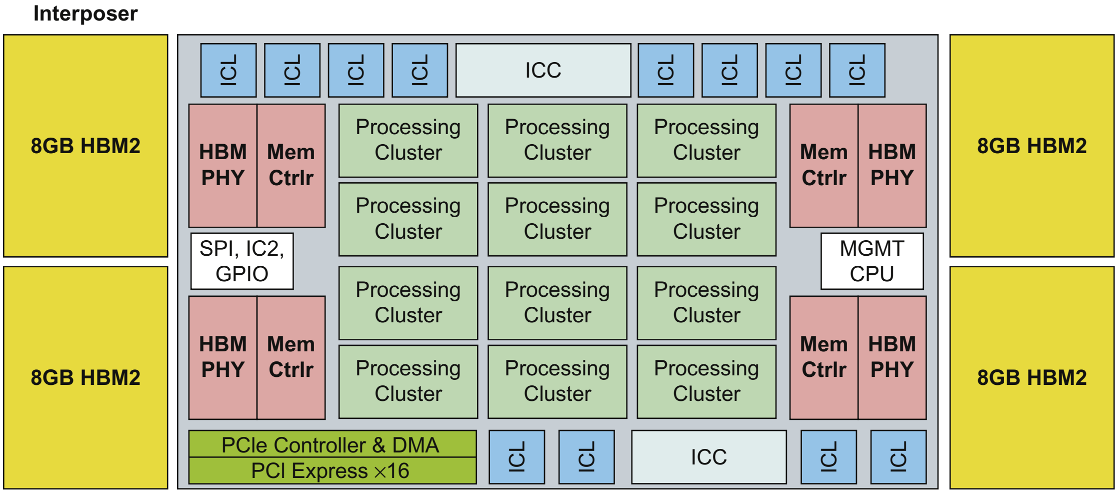


Figure 7.28 Block diagram of the Intel Lake Crest processor. Before being acquired by Intel, Crest said that the chip is almost a full reticle in TSMC 28 nm, which would make the die size 600–700 mm^2. This chip should be available in 2017. Intel is also building Knights Crest, which is a hybrid chip containing Xeon x86 cores and Crest accelerators.

Android operating system, and then potentially for Internet of Things (IoT) devices. It is a multicore design, supporting between 2 and 16 cores to deliver a desired cost-performance. It is designed either to be its own chip or to be part of a *system on a chip* (*SOC*). It has a much smaller area and energy budget than its TPU cousin. Figure 7.29 lists terms and acronyms found in this section.

Pixel Visual Core is an example of a new class of domain specific architectures for vision processing that we call *image processing units* (*IPUs*). IPUs solve the inverse problem of GPUs: they analyze and modify an input image in contrast to generating an output image. We call them IPUs to signal that, as a DSA, they do not need to do everything well because there will also be CPUs (and GPUs) in the system to perform non-input-vision tasks. IPUs rely on stencil computations mentioned above for CNNs.

The innovations of Pixel Visual Core include replacing the one-dimensional SIMD unit of CPUs with a two-dimensional array of processing elements (PEs). They provide a two-dimensional shifting network for the PEs that is aware of the two-dimensional spatial relationship between the elements, and a two-dimensional version of buffers that reduces accesses to off-chip memory. This novel hardware makes it easy to perform stencil computations that are central to both vision processing and CNN algorithms.

ISPs, the Hardwired Predecessors of IPUs

Most portable mobile devices (PMDs) have multiple cameras for input, which has led to hardwired accelerators called *image signal processors* (*ISPs*) for enhancing

Term	Acronym	Short explanation
Core	–	A processor. Pixel Visual Core can have 2–16 cores. The first implementation has 8; also called *stencil processor (STP)*
Halide	–	A domain-specific programming language for image processing that separates the algorithm from its execution schedule
Halo	–	An extended region around the 16 × 16 computation array to handle stencil computation near the borders of the array. It holds values, but doesn't compute
Image signal processors	ISP	A fixed function ASIC that improves the visual quality of an image; found in virtually all PMDs with cameras
Image processing unit	IPU	A DSA that solves the inverse problem of a GPU: it analyzes and modifies an *input* image in contrast to generating an *output* image
Line buffer pool	LB	A line buffer is designed to capture a sufficient number of full lines of an intermediate image to keep the next stage busy. Pixel Visual Core uses two-dimensional line buffers, each Change 64 to 128 KiB. The *Line Buffer Pool* contains one LB per core plus one LB for DMA
Network on chip	NOC	The network that connects the cores in Pixel Visual Core
Physical ISA	pISA	The Pixel Visual Core instruction set architecture (ISA) that is executed by the hardware
Processing element array	–	The 16 × 16 array of Processing Elements plus the halo that performs the 16-bit multiply-add operations. Each Processing Element includes a Vector Lane and local memory. It can shift data en mass to neighbors in any of four directions
Sheet generator	SHG	Does memory accesses of blocks of 1 × 1 to 31 × 31 pixels, which are called *sheets*. The different sizes allow the option of including the space for the halo or not
Scalar lane	SCL	Same operations as the Vector Lane except it adds instructions that handle jumps, branches, and interrupts, controls instruction flow to the vector array, and schedules all the loads and stores for the sheet generator. It also has a small instruction memory. It plays the same role as the scalar processor in a vector architecture
Vector lane	VL	Portion of the Processing Element that performs the computer arithmetic
Virtual ISA	vISA	The Pixel Visual Core ISA generated by the compiler. It is mapped to pISA before execution

Figure 7.29 A handy guide to Pixel Visual Core terms in Section 7.7. Figure 7.4 on page 437 has a guide for Sections 7.3–7.6.

input images. The ISP is usually a fixed function ASIC. Virtually every PMD today includes an ISP.

Figure 7.30 shows a typical organization of an image-processing system, including the lens, sensor, ISP, CPU, DRAM, and display. The ISP receives images, removes artifacts in images from the lens and the sensor, interpolates missing colors, and significantly improves the overall visual quality of the image. PMDs tend to have small lens and thus tiny noisy pixels, so this step is critical to producing high-quality photos and videos.

An ISP processes the input image in raster scan order by calculating a series of cascading algorithms via software configurable hardware building blocks, typically organized as a pipeline to minimize memory traffic. At each stage of the pipeline and for each clock cycle, a few pixels are input, and a few are output. Computation is typically performed over small neighborhoods of pixels (*stencils*). Stages are connected by buffers called *line buffers*. The line buffers help

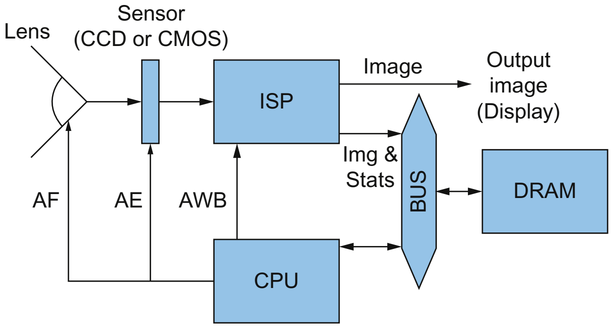


Figure 7.30 Diagram showing interconnection of the Image Signal Processor (ISP), CPU, DRAM, lens, and sensor. The ISP sends statistics to the CPU as well as the improved image either to the display or to DRAM for storage or later processing. The CPU then processes the image statistics and sends information to let the system adapt: *Auto White Balance* (AWB) to the ISP, *Auto Exposure* (AE) to the sensor, and *Auto Focus* (AF) to the lens, known as the *3As*.

keep the processing stages utilized via spatial locality by capturing just enough full lines of an intermediate image to facilitate the computation required by the next stage.

The enhanced image is either sent to a display or to DRAM for storage or for later processing. The ISP also sends statistics about the image (e.g., color and luma histograms, sharpness, and so on) to the CPU, which in turn it processes and sends information to help the system adapt.

Although efficient, ISPs have two major downsides. Given the increasing demand for improved image quality in handheld devices, the first is the inflexibility of an ISP, especially as it takes years to design and manufacture a new ISP within an SOC. The second is that these computing resources can be used only for the image-enhancing function, no matter what is needed at the time on the PMD. Current generation ISPs handle workloads at up to 2 Tera-operations per second on a PMD power budget, so a DSA replacement has to achieve similar performance and efficiency.

Pixel Visual Core Software

Pixel Visual Core generalized the typical hardwired pipeline organization of kernels of an ISP into a *directed acyclic graph* (*DAG*) of kernels. Pixel Visual Core image-processing programs are typically written in Halide, which is a domain-specific functional programming language for image processing. Figure 7.31 is a Halide example that blurs an image. Halide has a functional section to express the function being programmed and a separate schedule section to specify how to optimize that function to the underlying hardware.

```
Func buildBlur(Func input) {
      // Functional portion (independent of target processor)
    Func blur_x("blur_x"), blur_y("blur_y");
    blur_x(x,y) = (input(x-1,y) + input(x,y)*2 + input(x+1,y)) / 4;
    blur_y(x,y) = (blur_x(x,y-1) + blur_x(x,y)*2 + blur_x(x,y+1)) / 4;

    if (has_ipu) {
      // Schedule portion (directs how to optimize for target processor)
      blur_x.ipu(x,y);
      blur_y.ipu(x,y);
}
    return blur_y;
}
```

Figure 7.31 Portion of a Halide example to blur an image. The `ipu(x,y)` suffix schedules the function to Pixel Visual Core. A blur has the effect of looking at the image through a translucent screen, which reduces noise and detail. A Gaussian function is often used to blur the image.

Pixel Visual Core Architecture Philosophy

The power budget of PMDs is 6–8 W for bursts of 10–20 seconds, dropping down to tens of milliwatts when the screen is off. Given the challenging energy goals of a PMD chip, the Pixel Visual Core architecture was strongly shaped by the relative energy costs for the primitive operations mentioned in Chapter 1 and made explicit in Figure 7.32. Strikingly, a single 8-bit DRAM access takes as much energy as 12,500 8-bit additions or 7–100 8-bit SRAM accesses, depending on the organization of the SRAM. The 22 × to 150 × higher cost of IEEE 754 floating-point operations over 8-bit integer operations, plus the die size and energy benefits of storing narrower data, strongly favor using narrow integers whenever algorithms can accommodate them.

In addition to the guidelines from Section 7.2, these observations led to other themes that guided the Pixel Visual Core design:

- *Two-dimensional is better than one-dimensional*: Two-dimensional organizations can be beneficial for processing images as it minimizes communication distance and because the two- and three-dimensional nature of image data can utilize such organizations.
- *Closer is better than farther*: Moving data is expensive. Moreover, the relative cost of moving data to an ALU operation is increasing. And of course DRAM time and energy costs far exceed any local data storage or movement.

A primary goal in going from an ISP to an IPU is to get more reuse of the hardware via programmability. Here are the three main features of the Pixel Visual Core:

Operation	*Energy (pJ)*	Operation	Energy (pJ)	Operation	Energy (pJ)
8b DRAM LPDDR3	125.00	8b SRAM	1.2–17.1	16b SRAM	2.4–34.2
32b Fl. Pt. muladd	2.70	8b int muladd	0.12	16b int muladd	0.43
32b Fl. Pt. add	1.50	8b int add	0.01	16b int add	0.02

Figure 7.32 Relative energy costs per operation in picoJoules assuming TSMC 28-nm HPM process, which was the process Pixel Visual Core used [17][18][19][20]. The absolute energy cost are less than in Figure 7.2 because of using 28 nm instead of 90 nm, but the relative energy costs are similarly high.

1. Following the theme that two-dimensional is better than one-dimensional, Pixel Visual Core uses a two-dimensional SIMD architecture instead of one-dimensional SIMD architecture. Thus it has a two-dimensional array of independent *processing element*s (*PEs*), each of which contains 2 16-bit ALUs, 1 16-bit MAC unit, 10 16-bit registers, and 10 1-bit predicate registers. The 16-bit arithmetic follows the guideline of providing only the precision needed by the domain.
2. Pixel Visual Core needs temporary storage at each PE. Following the guideline from Section 7.2 of avoiding caches, this PE memory is a compiler-managed scratchpad memory. The logical size of each PE memory is 128 entries of 16 bits, or just 256 bytes. Because it would be inefficient to implement a separate small SRAM in each PE, Pixel Visual Core instead groups the PE memory of 8 PEs together in a single wide SRAM block. Because the PEs operate in SIMD fashion, Pixel Visual Core can bind all the individual reads and writes together to form a "squarer" SRAM, which is more efficient than narrow and deep or wide and shallow SRAMs. Figure 7.33 shows four PEs.
3. To be able to perform simultaneous stencil computations in all PEs, Pixel Visual Core needs to collect inputs from nearest neighbors. This communication pattern requires a "NSEW" (North, South, East, West) shift network: it can shift data en masse between the PEs in any compass direction. So that it doesn't lose pixels along the edges as it shifts images, Pixel Visual Core connects the endpoints of the network together to form a torus.

Note that the shift network is in contrast with the systolic array of processing element arrays in the TPU and Catapult. In this case, software explicitly moves the data in the desired direction across the array, whereas the systolic approach is a hardware-controlled, two-dimensional pipeline that moves data as a wavefront that is invisible to the software.

The Pixel Visual Core Halo

A 3×3, 5×5, or 7×7 stencil is going to get inputs from 1, 2, or 3 external pixels at the edges of the two-dimensional subset being computed (half of the dimension of the stencil minus one-half). That leaves two choices. Either Pixel Visual Core

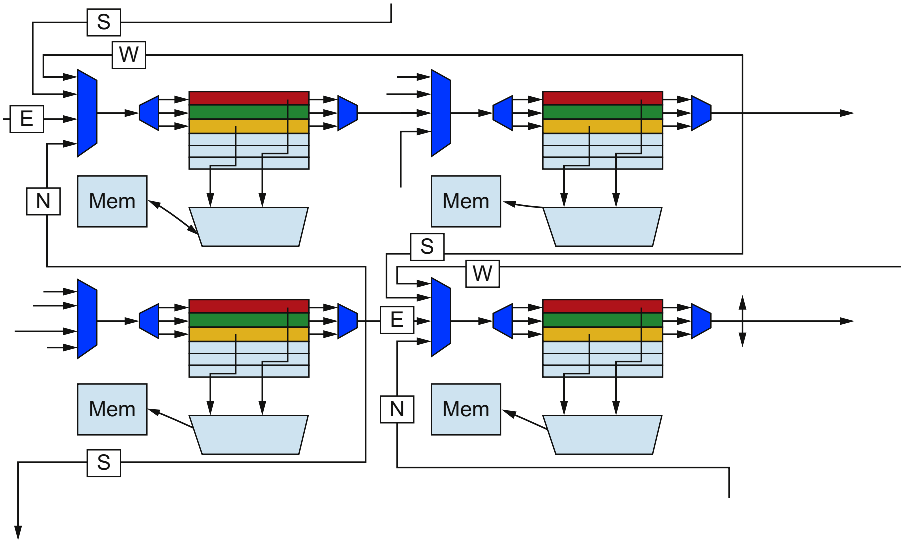


Figure 7.33 The two-dimensional SIMD includes two-dimensional shifting "N," "S," "E," "W," show the direction of the shift (North, South, East, West). Each PE has a software-controlled scratchpad memory.

under utilizes the hardware in the elements near the border, because they only pass input values, or Pixel Visual Core slightly extends the two-dimensional PEs with simplified PEs that leave out the ALUs. Because the difference in size between a standard PE and a simplified PE is about 2.2 ×, Pixel Visual Core has an extended array. This extended region is called the *halo*. Figure 7.34 shows two rows of a halo surrounding an 8x8 PE array and illustrates how an example 5 × 5 stencil computation in the upper-left corner relies on the halo.

A Processor of the Pixel Visual Core

The collection of 16 × 16 PEs and 4 halo lanes in each dimension, called the *PE array* or *vector array*, is the main computation unit of the Pixel Visual Core. It also has a load-store unit called a *Sheet Generator* (*SHG*). SHG refers to memory accesses of blocks of 1 × 1 to 256 × 256 pixels, which are called *sheets*. This happens during downsampling, and typical values are 16 × 16 or 20 × 20.

An implementation of Pixel Visual Core can have any even number of 2 or more cores, depending on the resources available. Thus it needs a network to connect them together, so every core also has an interface to the Network on Chip (NOC). A typical NOC implementation for Pixel Visual Core will not be an expensive cross switch, however, because those require data to travel a long distance, which is expensive. Leveraging the pipeline nature of the application, the NOC typically needs to communicate only to neighboring cores. It is implemented as a two-dimensional mesh, which allows power gating of pairs of cores under software control.

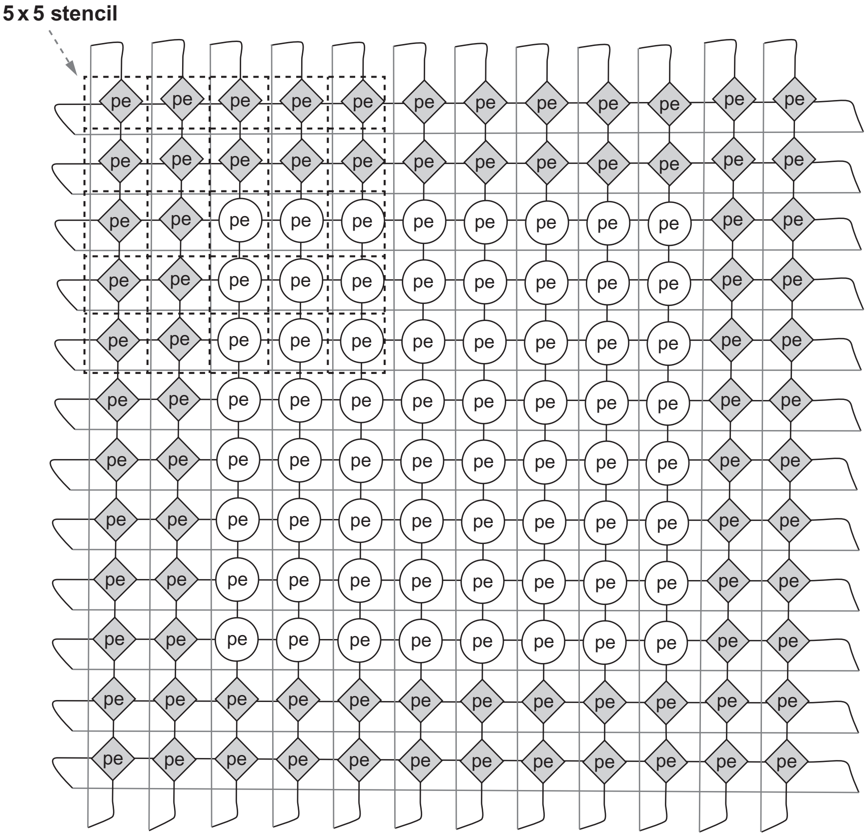


Figure 7.34 The two-dimensional array of full processing elements *(shown as unshaded circles)* surrounded by two layers of simplified processing elements *(shaded diamonds)* called a *halo*. In this figure, there are 8 × 8 or 64 full PEs with 80 simplified PEs in the halo. (Pixel Visual Core actually has 16 × 16 or 256 full PEs and two layers in its halo and thus 144 simplified PEs.) The edges of the halo are connected *(shown as gray lines)* to form a torus. Pixel Visual Core does a series of two-dimensional shifts across all processing elements to move the neighbor portions of each stencil computation into the center PE of the stencil. An example 5 × 5 stencil is shown in the upper-left corner. Note that 16 of the 25 pieces of data for this 5 × 5 stencil location come from halo processing elements.

Finally, the Pixel Visual Core also includes a scalar processor that is called a *scalar lane* (*SCL*). It is identical to the vector lane, except it adds instructions that handle jumps, branches, and interrupts, controls instruction flow to the vector array, and schedules all the loads and stores for the sheet generator. It also has a small instruction memory. Note that Pixel Visual Core has a single instruction stream that controls the scalar and vector units, similar to how a CPU core has a single instruction stream for its scalar and SIMD units.

In addition to cores, there is also a DMA engine to transfer data between DRAM and the line buffers while efficiently converting between image memory layout formats (e.g., packing/unpacking). As well as sequential DRAM accesses, the DMA engines perform vector-like gather reads of DRAM as well as sequential and strided reads and writes.

Pixel Visual Core Instruction Set Architecture

Like GPUs, Pixel Visual Core uses a two-step compilation process. The first step is compiling the programs from the target language (e.g., Halide) into *vISA* instructions. The Pixel Visual Core *vISA* (*virtual Instruction Set Architecture*) is inspired in part by the RISC-V instruction set, but it uses an image-specific memory model and extends the instruction set to handle image processing, and in particular, the two-dimensional notion of images. In vISA, the two-dimensional array of a core is infinite, the number of register is unbounded, and memory size is similarly unlimited. vISA instructions contain pure functions that don't directly access DRAM (see Figure 7.36), which greatly simplifies mapping them onto the hardware.

The next step is to compile the vISA program into a *pISA* (*physical Instruction Set Architecture*) program. Using vISA as the target of compilers allows the processor to be software-compatible with past programs and yet accept changes to the pISA instruction set, so vISA plays the same role that PTX does for GPUs (see Chapter 4).

Lowering from vISA to pISA takes two steps: compilation and mapping with early-bound parameters, and then patching the code with late-bound parameters. The parameters that must be bound include STP size, halo size, number of STPs, mapping of line buffers, mapping of kernels to processors, as well as register and local memory allocations.

Figure 7.35 shows that pISA is a very long instruction word (VLIW) instruction set with 119-bit-wide instructions. The first 43-bit field is for the Scalar Lane, the next 38-bit field specifies the computation by the two-dimensional PE array, and the third 12-bit field specifies the memory accesses by the two-dimensional PE array. The last two fields are immediates for computation or addressing. The operations for all the VLIW fields are what you'd expect: two's complement integer arithmetic, saturating integer arithmetic, logical operations, shifts, data transfers, and a few special ones like divide iteration and count leading zeros. The Scalar Lane supports a superset of the operations in the two-dimensional PE array, plus it adds instructions for control-flow and sheet-generator control. The 1-bit Predicate registers mentioned above enables conditional moves to registers (e.g., $A = B$ if C).

Field	Scalar	Math	Memory	Imm	MemImm
# Bits	43	38	12	16	10

Figure 7.35 VLIW format of the 119-bit pISA instruction.

Although the pISA VLIW instruction is very wide, Halide kernels are short, often just 200–600 instructions. Recall that as an IPU, it only needs to execute the compute-intensive portion of an application, leaving the rest of the functionality to CPUs and GPUs. Thus the instruction memory of a Pixel Visual Core holds just 2048 pISA instructions (28.5 KiB).

The Scalar Lane issues sheet generator instructions that access line buffers. Unlike other memory accesses within Pixel Visual Core, the latency can be more than 1 clock cycle, so they have a DMA-like interface. The lane first sets up the addresses and transfer size in special registers.

Pixel Visual Core Example

Figure 7.36 shows the vISA code that is output from the Halide compiler for the blur example in Figure 7.31, with comments added for clarity. It calculates a blur first in the *x* direction and then in the y direction using 16-bit arithmetic. The vISA code matches the functional part of the Halide program. This code can be thought of as executing across all the pixels of an image.

Pixel Visual Core Processing Element

One of the architectural decisions was how big to build the halo. Pixel Visual Core uses 16 × 16 PEs, and it adds a halo of 2 extra elements, so it can support 5 × 5

```
// vISA inner loop blur in x dimension
input.b16  t1 <- _input[x*1+(−1)][y*1+0][0]; // t1 = input[x−1,y]
input.b16  t2 <- _input[x*1+0][y*1+0][0];  // t2 = input[x,y]
mov.b16    st3 <- 2;
mul.b16    t4 <- t2, st3;  //t4 = input[x,y] * 2
add.b16    t5 <- t1, t4;  //t5 = input[x−1,y] + input[x,y]*2
input.b16  t6 <- _input[x*1+1][y*1+0][0];  // t6 = input[x+1,y]
add.b16    t7 <- t5, t6;  //t7 = input[x+1,y]+input[x,y]+input[x−1,y]*2
mov.b16    st8 <- 4;
div.b16    t9 <- t7, st8;  //t9 = t7/4
output.b16 _blur_x[x*1+0][y*1+0][0] <- t9; // blur_x[x,y] = t7/4
// vISA inner loop blur in y dimension
input.b16  t1 <- _blur_x[x*1+0][y*1+(−1)][0]; // t1 = blur_x[x,y−1]
input.b16  t2 <- _blur_x[x*1+0][y*1+0][0];  // t2 = blur_x[x,y]
mov.b16    st3 <- 2;
mul.b16    t4 <- t2, st3;  //t4 = blur_x[x,y] * 2
add.b16    t5 <- t1, t4;  //t5 = blur_x[x,y−1] + blur_x[x,y]*2
input.b16  t6 <- _blur_x[x*1+0][y*1+1][0];  // t6 = blur_x[x,y+1]
add.b16    t7 <- t5, t6;  //t7 = blurx[x,y+1]+blurx[x,y−1]+blurx[x,y]*2
mov.b16    st8 <- 4;
div.b16    t9 <- t7, st8;  //t9 = t7/4
output.b16 _blur_y[x*1+0][y*1+0][0] <- t9; // blur_y[x,y] = t7/4
```

Figure 7.36 Portion of the vISA instructions compiled from the Halide Blur code in Figure 7.31. This vISA code corresponds to the functional part of the Halide code.

stencils directly. Note that the bigger the array of PEs, the less the halo overhead to support a given stencil size.

For Pixel Visual Core, the smaller size of the halo PEs and the 16 × 16 arrays means it only costs 20% more area for the halo. For a 5 × 5 stencil, Pixel Visual Core can calculate 1.8 times as many results per clock cycle ($16^2/12^2$), and the ratio is 1.3 for a 3 × 3 stencil ($16^2/14^2$).

The design of the arithmetic unit of the PE is driven by multiply-accumulate (MAC), which is a primitive of stencil computation. Pixel Visual Core native MACs are 16-bits wide for the multiplies, but they can accumulate at a 32-bit width. Pipelining MAC would use energy unnecessarily because of the reading and writing of the added pipeline register. Thus the multiply-add hardware determines the clock cycle. The other operations, previously mentioned, are the traditional logical and arithmetic operations along with saturating versions of the arithmetic operations and a few specialized instructions.

The PE has two 16-bit ALUs that can operate in a variety of ways within a single clock cycle:

- Independently, producing two 16-bit results: A op B, C op D.
- Fused, producing just one 16-bit result: A op (C op D).
- Joined, producing one 32-bit result: A:C op B:D.

Two-Dimensional Line Buffers and Their Controller

Because DRAM accesses use so much energy (see Figure 7.32), the Pixel Visual Core memory system was carefully designed to minimize the number of DRAM accesses. The key innovation is the *two-dimensional line buffer*.

Kernels are logically running on separate cores, and they are connected in a DAG with input from the sensor or DRAM and output to DRAM. The line buffers hold portions of the image being calculated between kernels. Figure 7.37 shows the logical use of line buffers in Pixel Visual Core.

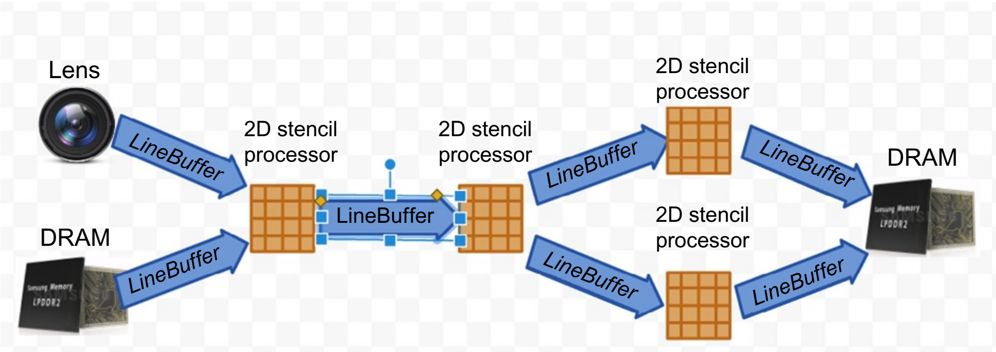


Figure 7.37 Programmer view of Pixel Visual Core: a directed-acyclic graph of kernels.

Here are four features that the two-dimensional line buffer must support:

1. It must support two-dimensional stencil computations of various sizes, which are unknown at design time.
2. Because of the halo, for the 16 × 16 PE array in Pixel Visual Core, the STPs will want to read 20 × 20 blocks of pixels from the line buffer and write 16 × 16 blocks of pixels to the line buffer. (As previously mentioned, they call these blocks of pixels *sheets*.)
3. Because the DAG is programmable, we need line buffers that can be allocated by software between any two cores.
4. Several cores may need to read data from the same line buffer. Thus a line buffer should support multiple consumers, although it needs just one producer.

Line buffers in Pixel Visual Core are really a multi-reader, two-dimensional FIFO abstraction on top of a relatively large amount of SRAM: 128 KiB per instance. It contains temporary "images" that are used just once, so a small, dedicated local FIFO is much more efficient than a cache for data in distant memory.

To accommodate the size mismatch between reading 20 × 20 blocks of pixels and writing 16 × 16 blocks, the fundamental unit of allocation in the FIFO is a group of 4 × 4 pixels. Per stencil processor, there is one *Line Buffer Pool* (*LBP*) that can have eight logical line buffers (*LB*), plus one LBP for DMA of I/O. The LBP has three levels of abstraction:

1. At the top, the LBP controller supports eight LBs as logical instances. Each LB has one FIFO producer and up to eight FIFO consumers per LB.
2. The controller keeps track of a set of head and tail pointers for each FIFO. Note that the sizes of the line buffers inside the LBP are flexible and up to the controller.
3. At the bottom are many physical memory banks to support the bandwidth requirements. Pixel Visual Core has eight physical memory banks, each having a 128-bit interface and 16 KiB of capacity.

The controller for the LBP is challenging because it must fulfill the bandwidth demands of the STPs and I/O DMAs as well as schedule all their reads and writes to the banks of physical SRAM memory. The LBP controller is one of the most complicated pieces of Pixel Visual Core.

Pixel Visual Core Implementation

The first implementation of Pixel Visual Core was as a separate chip. Figure 7.38 shows the floorplan of the chip, which has 8 cores. It was fabricated in a TSMC 28 nm technology in 2016. The chip dimensions are 6 × 7.2 mm, it runs at 426 MHz, it is stacked with 512 MB DRAM as Silicon in Package, and consumes

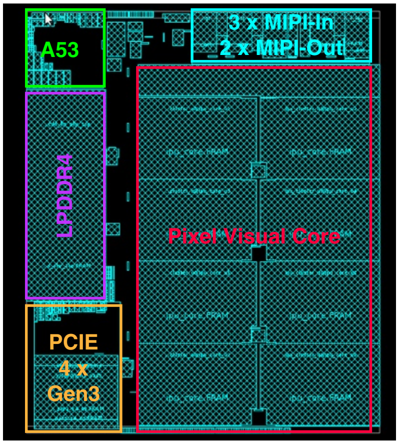


Figure 7.38 Floor plan of the 8-core Pixel Visual Core chip. A53 is an ARMv7 core. LPDDR4 is a DRAM controller. PCIE and MIPI are I/O buses.

(including the DRAM) 187–4500 mW depending on the workload. About 30% of the power for the chip is for an ARMv7 A53 core for control, the MIPI, the PCIe, the PCIe, and the LPDDR interfaces, interface is just over half this die at 23 mm^2. Power for Pixel Visual Core running a worst case "power virus" can go as high as 3200 mW. Figure 7.39 shows the floor plan of a core.

Summary: How Pixel Visual Core Follows the Guidelines

Pixel Visual Core is a multicore DSA for image and vision processing intended as a stand-alone chip or as an IP block for mobile device SOCs. As we will see in Section 7.9, its performance per watt for CNNs are factors of 25–100 better than CPUs and GPUs. Here is how the Pixel Visual core followed the guidelines in Section 7.2.

1. *Use dedicated memories to minimize the distance over which data is moved.*
   Perhaps the most distinguishing architecture feature of Pixel Visual Core is the software-controlled, two-dimensional line buffers. At 128 KiB per core, they are a significant fraction of the area. Each core also has 64 KiB of software-controlled PE memory for temporary storage.
2. *Invest the resources saved from dropping advanced microarchitectural optimizations into more arithmetic units or bigger memories.*
   Two other key features of Pixel Visual Core are a 16 × 16 two-dimensional array of processing elements per core and a two-dimensional shifting network between the processing elements. It offers a halo region that acts as a buffer to allow full utilization of its 256 arithmetic units.

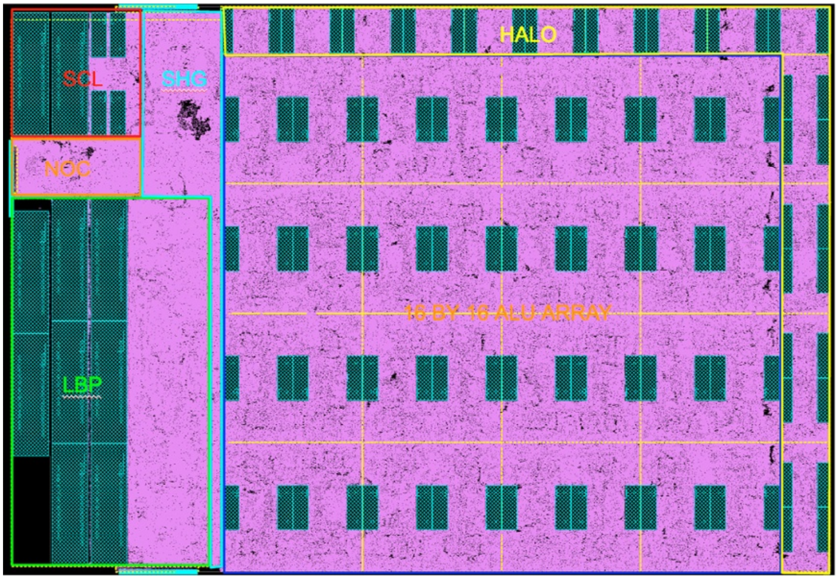


Figure 7.39 Floor plan of a Pixel Visual Core. From left to right, and top down: the scalar lane (SCL) is 4% of the core area, NOC is 2%, the line buffer pool (LBP) is 15%, the sheet generator (SHG) is 5%, the halo is 11%, and the processing element array is 62%. The torus connection of the halo makes each of the four edges of the array logical neighbors. It is more area-efficient to collapse the halo to just two sides, which preserves the topology.

3. *Use the easiest form of parallelism that matches the domain.*
   Pixel Visual Core relies on two-dimensional SIMD parallelism using its PE array, VLIW to express instruction-level parallelism, and multiple program multiple data (MPMD) parallelism to utilize multiple cores.
4. *Reduce data size and type to the simplest needed for the domain.*
   Pixel Visual Core relies primarily on 8-bit and 16-bit integers, but it also works with 32-bit integers, albeit more slowly.
5. *Use a domain-specific programming language to port code to the DSA.*
   Pixel Visual Core is programmed in the domain-specific language Halide for image processing and in TensorFlow for CNNs.

7.8 Cross-Cutting Issues

Heterogeneity and System on a Chip (SOC)

The easy way to incorporate DSAs into a system is over the I/O bus, which is the approach of the data center accelerators in this chapter. To avoid fetching memory operands over the slow I/O bus, these accelerators have local DRAM.

Amdahl's Law reminds us that the performance of an accelerator is limited by the frequency of shipping data between the host memory and the accelerator memory. There will surely be applications that would benefit from the host CPU and the accelerators to be integrated into the same *system on a chip* (*SOC*), which is one of the goals of Pixel Visual Core and eventually the Intel Crest.

Such a design is called an *IP block*, standing for *Intellectual Property*, but a more descriptive name might be portable design block. IP blocks are typically specified in a hardware description language like Verilog or VHDL to be integrated into the SOC. IP blocks enable a marketplace where many companies make IP blocks that other companies can buy to build the SOCs for their applications without having to design everything themselves. Figure 7.40 indicates the importance of IP blocks by plotting the number of IP blocks across generations of Apple PMD SOCs; they tripled in just four years. Another indication of the importance of IP blocks is that the CPU and GPU get only one-third of the area of the Apple SOCs, with IP blocks occupying the remainder (Shao and Brooks, 2015).

Designing an SOC is like city planning, where independent groups lobby for limited resources, and finding that the right compromise is difficult. CPUs, GPUs, caches, video encoders, and so on have adjustable designs that can shrink or expand to use more or less area and energy to deliver more or less performance. Budgets will differ depending on whether the SOC is for tablets or for IoT. Thus an IP block must be scalable in area, energy, and performance. Moreover, it is especially important for a new IP block to offer a small resource version because it may not already have a well-established foothold in the SOC ecosystem; adoption is much easier if the initial resource request can be modest. The Pixel Visual Core approach is a multicore design, allowing the SOC engineer to choose between 2 and 16 cores to match the area and power budget and desired performance.

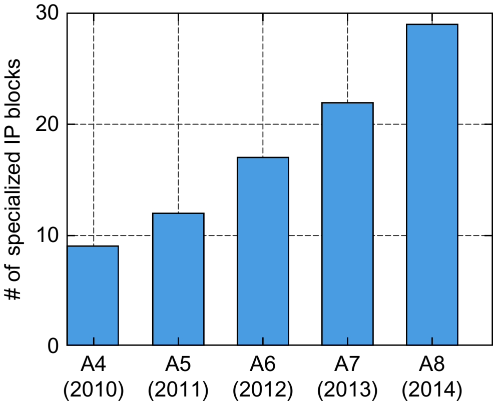


Figure 7.40 Number of IP blocks in Apple SOCs for the iPhone and iPad between 2010 and 2014 (Shao and Brooks, 2015).

It will be interesting to see whether the attractiveness of integration leads to most data center processors coming from traditional CPU companies with IP accelerators integrated into the CPU die, or whether systems companies will continue designing their own accelerators and include IP CPUs in their ASICs.

An Open Instruction Set

One challenge for designers of DSAs is determining how to collaborate with a CPU to run the rest of the application. If it's going to be on the same SOC, then a major decision is which CPU instruction set to choose, because until recently virtually every instruction set belonged to a single company. Previously, the practical first step of an SOC was to sign a contract with a company to lock in the instruction set.

The alternative was to design your own custom RISC processor and to port a compiler and libraries to it. The cost and hassle of licensing IP cores led to a surprisingly large number of do-it-yourself simple RISC processors in SOCs. One AMD engineer estimated that there were 12 instruction sets in a modern microprocessor!

RISC-V offers a third choice: a viable free and open instruction set with plenty of opcode space reserved for adding instructions for domain-specific coprocessors, which enables the previously mentioned tighter integration between CPUs and DSAs. SOC designers can now select a standard instruction set that comes with a large base of support software without having to sign a contract.

They still have to pick the instruction set early in the design, but they don't have to pick one company and sign a contract. They can design a RISC-V core themselves, they can buy one from the several companies that sell RISC-V IP blocks, or they can download one of the free open-source RISC-V IP blocks developed by others. The last case is analogous to open-source software, which offers web browsers, compilers, operating systems, and so on that volunteers maintain for users to download and use for free.

As a bonus, the open nature of the instruction set improves the business case for small companies offering RISC-V technology because customers don't have to worry about the long-term viability of a company with its own unique instruction set.

Another attraction of RISC-V for DSAs is that the instruction set is not as important as it is for general-purpose processors. If DSAs are programmed at higher levels using abstractions like DAGs or parallel patterns, as is the case for Halide and TensorFlow, then there is less to do at the instruction set level. Moreover, in a world where performance-cost and energy-cost advances come from adding DSAs, binary compatibility may not play as important a role as in the past.

At the time of this writing, the future of the open RISC-V instruction set appears promising. (We wish we could peer into the future and learn the status of RISC-V from now to the next edition of this book!)

7.9 Putting It All Together: CPUs Versus GPUs Versus DNN Accelerators

We now use the DNN domain to compare the cost-performance of the accelerators in this chapter.[2] We start with a thorough comparison of the TPU to standard CPUs and GPUs and then add brief comparisons to Catapult and Pixel Visual Core.

Figure 7.41 shows the six benchmarks we use in this comparison. They consist of two examples of each of the three types of DNNs in Section 7.3. These six benchmarks represent 95% of TPU inference workload in Google data centers in 2016. Typically written in TensorFlow, they are surprisingly short: just 100–1500 lines of code. They are small pieces of larger applications that run on the host server, which can be thousands to millions of lines of C++ code. The applications are typically user-facing, which leads to rigid response-time limits, as we will see.

Figures 7.42 and 7.43 show the chips and servers being compared. They are server-class computers deployed in Google data centers at the same time that TPUs were deployed. To be deployed in Google data centers, they must at least check for internal memory errors, which excluded some choices, such as the Nvidia Maxwell GPU. For Google to purchase and deploy them, the machines had to be sensibly configured, and not awkward artifacts assembled solely to win benchmarks.

The traditional CPU server is represented by an 18-core, dual-socket Haswell processor from Intel. This platform is also the host server for GPUs or TPUs.

		DNN layers							
Name	**LOC**	**FC**	**Conv**	**Element**	**Pool**	**Total**	**Weights**	**TPU Ops/Weight**	**% deployed TPUs 2016**
MLP0	100	5				5	20M	200	61%
MLP1	1000	4				4	5M	168	
LSTM0	1000	24		34		58	52M	64	29%
LSTM1	1500	37		19		56	34M	96	
CNN0	1000		16			16	8M	2888	5%
CNN1	1000	4	72		13	89	100M	1750	

Figure 7.41 Six DNN applications (two per DNN type) that represent 95% of the TPU's workload. The 10 columns are the DNN name; the number of lines of code; the types and number of layers in the DNN (FC is fully connected; Conv is convolution; Element is element-wise operation of LSTM, see Section 7.3; and Pool is pooling, which is a downsizing stage that replaces a group of elements with its average or maximum); the number of weights; TPU operational intensity; and TPU application popularity in 2016. The operational intensity varies between TPU, CPU, and GPU because the batch sizes vary. The TPU can have larger batch sizes while still staying under the response time limit. One DNN is RankBrain (Clark, 2015), one LSTM is GNM Translate (Wu et al., 2016), and one CNN is DeepMind AlphaGo (Silver et al., 2016; Jouppi, 2016).

---

[2] This section is also largely based upon the paper "In-Datacenter Performance Analysis of a Tensor Processing Unit" Jouppi et al., 2017, of which one of your book authors was a coauthor.

					Measured		TOPS/s			
Chip model	**mm^2**	**nm**	**MHz**	**TDP**	**Idle**	**Busy**	**8b**	**FP**	**GB/s**	**On-chip memory**
Intel Haswell	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB
NVIDIA K80	561	28	560	150W	25W	98W	–	2.8	160	8 MiB
TPU	<331*	28	700	75W	28W	40W	92	–	34	28 MiB

*The TPU die size is less than half of the Haswell die size.

Figure 7.42 The chips used by the benchmarked servers are Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors.

				Measured power	
Server	**Dies/Server**	**DRAM**	**TDP**	**Idle**	**Busy**
Intel Haswell	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	8	256 GiB (host) + 12 GiB × 8	1838W	357W	991W
TPU	4	256 GiB (host) + 8 GiB × 4	861W	290W	384W

Figure 7.43 Benchmarked servers that use the chips in Figure 7.42. The low-power TPU allows for better rack-level density than the high-power GPU. The 8 GiB DRAM per TPU is Weight Memory.

Haswell is fabricated in an Intel 22-nm process. Both the CPU and GPU are very large dies: about 600 mm^2!

The GPU accelerator is the Nvidia K80. Each K80 card contains two dies and offers SECDED on internal memory and DRAM. Nvidia states that (Nvidia, 2016)

> the K80 Accelerator dramatically lowers datacenter cost by delivering application performance with fewer, more powerful servers.

DNN researchers frequently used K80s in 2015, which is when they were deployed at Google. Note that K80s were also chosen for new cloud-based GPUs by Amazon Web Services and by Microsoft Azure in late 2016.

Because the number of dies per benchmarked server varies between 2 and 8, the following figures show results normalized per die, except for Figure 7.50, which compares the performance/watt of whole servers.

Performance: Rooflines, Response Time, and Throughput

To illustrate the performance of the six benchmarks on the three processors, we adapt the Roofline performance model in Chapter 4. To use the Roofline model for the TPU, when DNN applications are quantized, we first replace floating-point operations with integer multiply-accumulate operations. As weights do not normally fit in on-chip memory for DNN applications, the second change is to redefine operational intensity to be integer operations per byte of weights read (Figure 7.41).

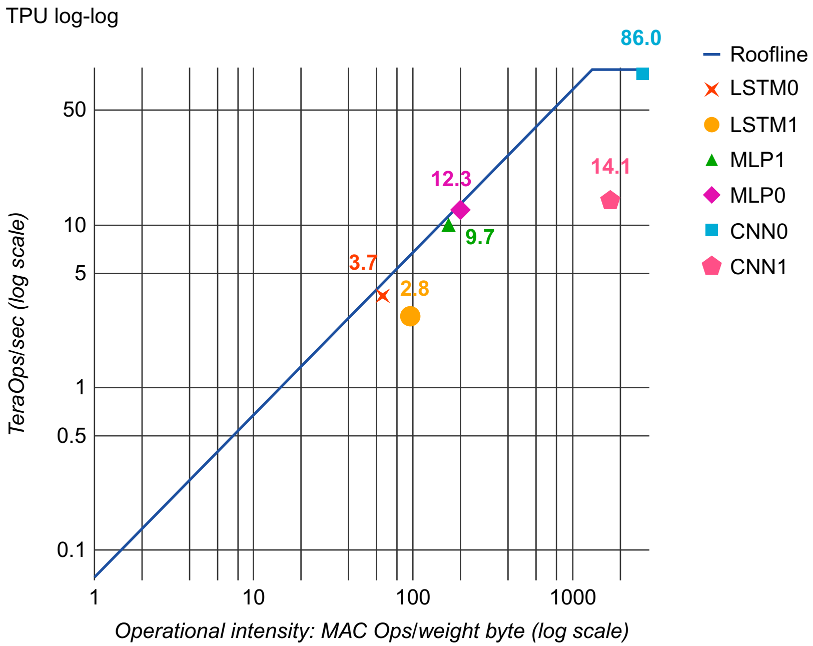


Figure 7.44 TPU Roofline. Its ridge point is far to the right at 1350 multiply-accumulate operations per byte of weight memory. CNN1 is much further below its Roofline than the other DNNs because it spends about a third of the time waiting for weights to be loaded into the matrix unit and because the shallow depth of some layers in the CNN results in only half of the elements within the matrix unit holding useful values (Jouppi et al., 2017).

Figure 7.44 shows the Roofline model for a single TPU on log-log scales. The TPU has a long "slanted" part of its Roofline, where operational intensity means that performance is limited by memory bandwidth rather than by peak compute. Five of the six applications are happily bumping their heads against the ceiling: the MLPs and LSTMs are memory-bound, and the CNNs are computation-bound. The single DNN that is not bumping its head against the ceiling is CNN1. Despite CNNs having very high operational intensity, CNN1 is running at only 14.1 Tera Operations Per Second (TOPS), while CNN0 runs at a satisfying 86 TOPS.

For readers interested into a deep dive into what happened with CNN1, Figure 7.45 uses performance counters to give partial visibility into the utilization of the TPU. The TPU spends less than half of its cycles performing matrix operations for CNN1 (column 7, row 1). On each of those active cycles, only about half of the 65,536 MACs hold useful weights because some layers in CNN1 have shallow feature depths. About 35% of cycles are spent waiting for weights to load from memory into the matrix unit, which occurs during the four fully connected layers that run at an operational intensity of just 32. This leaves roughly 19% of cycles not

Application	MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1	Mean	Row
Array active cycles	12.7%	10.6%	8.2%	10.5%	78.2%	46.2%	28%	1
Useful MACs in 64K matrix (% peak)	12.5%	9.4%	8.2%	6.3%	78.2%	22.5%	23%	2
Unused MACs	0.3%	1.2%	0.0%	4.2%	0.0%	23.7%	5%	3
Weight stall cycles	53.9%	44.2%	58.1%	62.1%	0.0%	28.1%	43%	4
Weight shift cycles	15.9%	13.4%	15.8%	17.1%	0.0%	7.0%	12%	5
Non-matrix cycles	17.5%	31.9%	17.9%	10.3%	21.8%	18.7%	20%	6
RAW stalls	3.3%	8.4%	14.6%	10.6%	3.5%	22.8%	11%	7
Input data stalls	6.1%	8.8%	5.1%	2.4%	3.4%	0.6%	4%	8
TeraOp/s (92 Peak)	12.3	9.7	3.7	2.8	86.0	14.1	21.4	9

Figure 7.45 Factors limiting TPU performance of the NN workload based on hardware performance counters. Rows 1, 4, 5, and 6 total 100% and are based on measurements of activity of the matrix unit. Rows 2 and 3 further break down the fraction of 64K weights in the matrix unit that hold useful weights on active cycles. Our counters cannot exactly explain the time when the matrix unit is idle in row 6; rows 7 and 8 show counters for two possible reasons, including RAW pipeline hazards and PCIe input stalls. Row 9 (TOPS) is based on measurements of production code while the other rows are based on performance-counter measurements, so they are not perfectly consistent. Host server overhead is excluded here. The MLPs and LSTMs are memory-bandwidth limited, but CNNs are not. CNN1 results are explained in the text.

explained by the matrix-related counters. Because of overlapped execution on the TPU, we do not have exact accounting for those cycles, but we can see that 23% of cycles have stalls for RAW dependences in the pipeline and that 1% are spent stalled for input over the PCIe bus.

Figures 7.46 and 7.47 show Rooflines for Haswell and the K80. The six NN applications are generally further below their ceilings than the TPU in Figure 7.44. Response-time limits are the reason. Many of these DNN applications are parts of services that are part of end-user-facing services. Researchers have demonstrated that small increases in response time cause customers to use a service less (see Chapter 6). Thus, although training may not have hard response-time deadlines, inference usually does. That is, inference cares about throughput only while it is maintaining the latency bound.

Figure 7.48 illustrates the impact of the 99th percentile response-time limit of 7 ms for MLP0 on Haswell and the K80, which was required by the application developer. (The inferences per second and 7-ms latency include the server host time as well as the accelerator time.) They can operate at 42% and 37%, respectively, with the highest throughput achievable for MLP0, if the response-time limit is relaxed. Thus, although CPUs and GPUs have potentially much higher throughput, it's wasted if they don't meet the response-time limit. These bounds affect the TPU as well, but at 80% in Figure 7.48, it is operating much closer to its highest MLP0 throughput. As compared with CPUs and GPUs, the single-threaded TPU has none of the sophisticated microarchitectural features discussed in Section 7.1 that consume transistors and energy to improve the average case but not the 99th-percentile case.

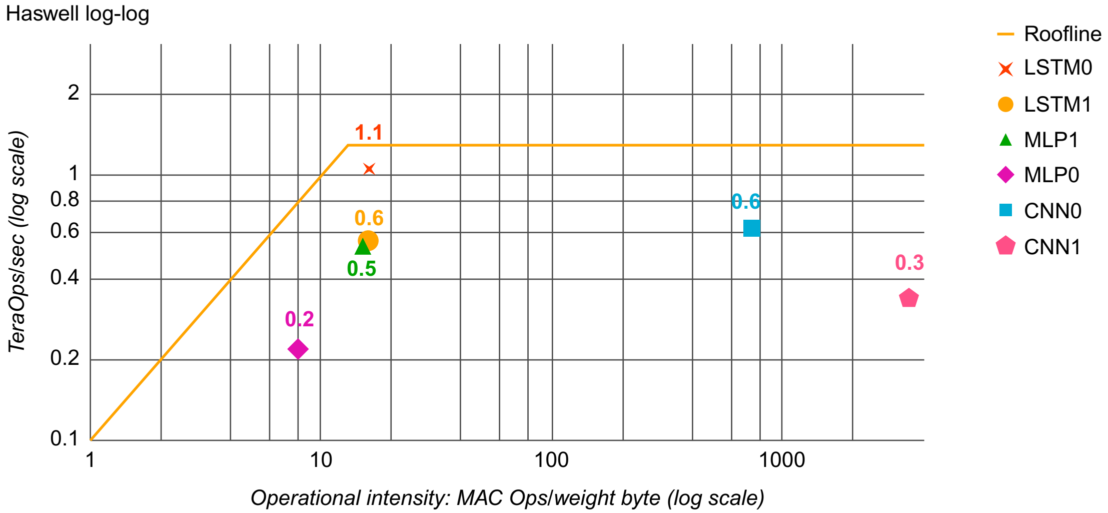


Figure 7.46 Intel Haswell CPU Roofline with its ridge point at 13 multiply-accumulate operations/byte, which is much further to the left than in Figure 7.44.

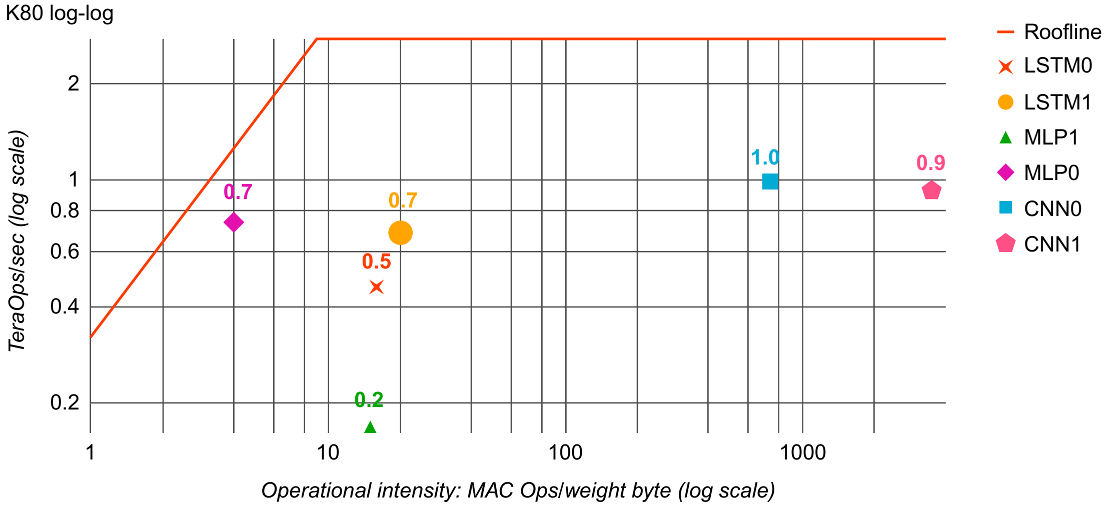


Figure 7.47 NVIDIA K80 GPU die Roofline. The much higher memory bandwidth moves the ridge point to 9 multiply-accumulate operations per weight byte, which is even further to the left than in Figure 7.46.

Figure 7.49 gives the bottom line of relative inference performance per die, including the host server overhead for the two accelerators. Recall that architects use the geometric mean when they don't know the actual mix of programs that will be run. For this comparison, however, we *do* know the mix (Figure 7.41). The

Type	Batch	99th% response	Inf/s (IPS)	% max IPS
CPU	16	7.2 ms	5482	42%
CPU	64	21.3 ms	13,194	100%
GPU	16	6.7 ms	13,461	37%
GPU	64	8.3 ms	36,465	100%
TPU	200	7.0 ms	225,000	80%
TPU	250	10.0 ms	280,000	100%

Figure 7.48 99th% response time and per die throughput (IPS) for MLP0 as batch size varies. The longest allowable latency is 7 ms. For the GPU and TPU, the maximum MLP0 throughput is limited by the host server overhead.

Type	MLP0	MLP1	LSTM0	LSTM1	CNN0	CNN1	Mean
GPU	2.5	0.3	0.4	1.2	1.6	2.7	1.9
TPU	41.0	18.5	3.5	1.2	40.3	71.0	29.2
Ratio	16.7	60.0	8.0	1.0	25.4	26.3	15.3

Figure 7.49 K80 GPU and TPU performance relative to CPU for the DNN workload. The mean uses the actual mix of the six applications in Figure 7.41. Relative performance for the GPU and TPU includes host server overhead. Figure 7.48 corresponds to the second column of this table (MLP0), showing relative IPS that meet the 7-ms latency threshold.

weighted mean in the last column of Figure 7.49 using the actual mix makes the GPU up to 1.9 times, and the TPU is 29.2 times as fast as the CPU, so the TPU is 15.3 times as fast as the GPU.

Cost-Performance, TCO, and Performance/Watt

When buying computers by the thousands, cost-performance trumps general performance. The best cost metric in a data center is total cost of ownership (TCO). The actual price Google pays for thousands of chips depends on negotiations between the companies involved. For business reasons, Google can't publish such price information or data that might let them be deduced. However, power is correlated with TCO, and Google can publish watts per server, so we use performance/watt as our proxy for performance/TCO. In this section, we compare servers (Figure 7.43) rather than single dies (Figure 7.42).

Figure 7.50 shows the weighted mean performance/watt for the K80 GPU and TPU relative to the Haswell CPU. We present two different calculations of performance/watt. The first ("total") includes the power consumed by the host CPU server when calculating performance/watt for the GPU and TPU. The second ("incremental") subtracts the host CPU server power from the total for the GPU and TPU beforehand.

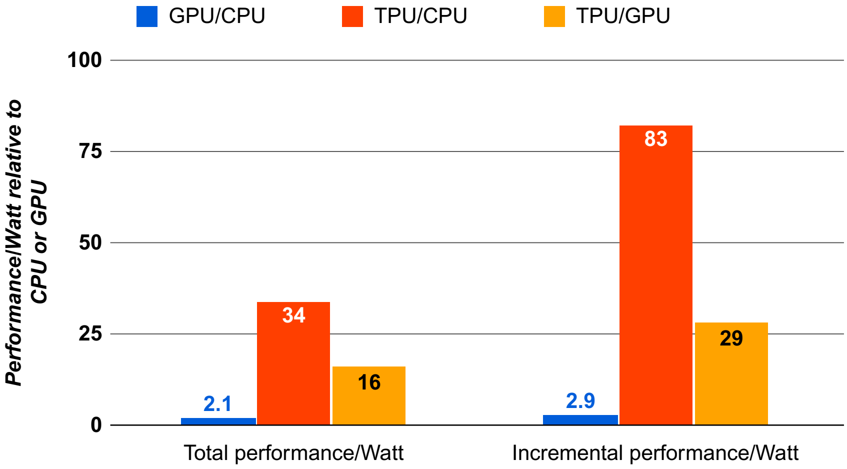


Figure 7.50 Relative performance/watt of GPU and TPU servers to CPU or GPU servers. Total performance/watt includes host server power, but incremental doesn't. It is a widely quoted metric, but we use it as a proxy for performance/TCO in the data center.

For total-performance/watt, the K80 server is 2.1 × Haswell. For incremental-performance/watt, when Haswell power is omitted, the K80 server is 2.9 ×.

The TPU server has 34 times better total-performance/watt than Haswell, which makes the TPU server 16 times the performance/watt of the K80 server. The relative incremental-performance/watt—which was Google's justification for a custom ASIC—is 83 for the TPU, which lifts the TPU to 29 times the performance/watt of the GPU.

Evaluating Catapult and Pixel Visual Core

Catapult V1 runs CNNs 2.3 × as fast as a 2.1 GHz, 16-core, dual-socket server (Ovtcharov et al., 2015a). Using the next generation of FPGAs (14-nm Arria 10), performance goes up 7 ×, and perhaps even 17 × with more careful floorplanning and scaling up of the Processing Elements (Ovtcharov et al., 2015b). In both cases, Catapult increases power by less than 1.2 ×. Although it's apples versus oranges, the TPU runs its CNNs 40 × to 70 × versus a somewhat faster server (see Figures 7.42, 7.43, and 7.49).

Because Pixel Visual Core and the TPU are both made by Google, the good news is that we can directly compare performance for CNN1, which is a common DNN, although it had to be translated from TensorFlow. It runs with batch size of 1 instead of 32 as in the TPU. The TPU runs CNN1 about 50 times as fast as Pixel Visual Core, which makes Pixel Visual Core about half as fast as the GPU and a little faster than Haswell. Incremental performance/watt for CNN1 raises Pixel Visual Core to about half the TPU, 25 times the GPU, and 100 times the CPU.

Because the Intel Crest is designed for training rather than inference, it wouldn't be fair to include it in this section, even if it were available to measure.

7.10 Fallacies and Pitfalls

In these early days of both DSAs and DNNs, fallacies abound.

Fallacy *It costs $100 million to design a custom chip.*

Figure 7.51 shows a graph from an article that debunks the widely quoted $100-million myth that it was "only" $50 million, with most of the cost being salaries (Olofsson, 2011). Note that the author's estimate is for sophisticated processors that include features that DSAs by definition omit, so even if there were no improvement to the development process, you would expect the cost of a DSA design to be less.

Why are we more optimistic six years later, when, if anything, mask costs are even higher for the smaller process technologies?

First, software is the largest category, at almost a third of the cost. The availability of applications written in domain-specific languages allows the compilers to do most of the work of porting the applications to your DSA, as we saw for the TPU and Pixel Visual Core. The open RISC-V instruction set will also help reduce the cost of getting system software as well as cut the large IP costs.

Mask and fabrication costs can be saved by having multiple projects share a single reticle. As long as you have a small chip, amazingly enough, for $30,000 anyone can get 100 untested parts in 28-nm TSMC technology (Patterson and Nikolić, 2015).

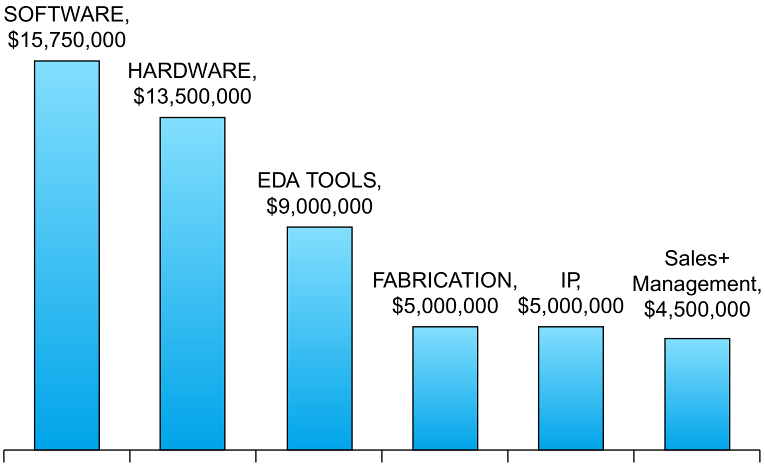


Figure 7.51 The breakdown of the $50 million cost of a custom ASIC that came from surveying others (Olofsson, 2011). The author wrote that his company spent just $2 million for its ASIC.

Perhaps the biggest change is to hardware engineering, which is more than a quarter of the cost. Hardware engineers have begun to follow their software colleagues to use agile development. The traditional hardware process not only has separate phases for design requirements, architecture, logical design, layout, verification, and so on, but also it uses different job titles for the people who perform each of the phases. This process is heavy on planning, documentation, and scheduling in part because of the change in personnel each phase.

Software used to follow this "waterfall" model as well, but projects were so commonly late, over budget, and even canceled that it led to a radically different approach. The Agile Manifesto in 2001 basically said that it was much more likely that a small team that iterated on an incomplete but working prototype shown regularly to customers would produce useful software on schedule and on budget more than the traditional plan-and-document approach of the waterfall process would.

Small hardware teams now do agile iterations (Lee et al., 2016). To ameliorate the long latency of a chip fabrication, engineers do some iterations using FPGAs because modern design systems can produce both the EDIF for FPGAs and chip layout from a single design. FPGA prototypes run 10–20 times slower than chips, but that is still much faster than simulators. They also do "tape-in" iterations, where you do all the work of a tape-out for your working but incomplete prototype, but you don't pay the costs of fabricating a chip.

In addition to an improved development process, more modern hardware design languages to support them (Bachrach et al., 2012), and advances in automatic generation of hardware from high-level domain-specific languages (Canis et al., 2013; Huang et al., 2016; Prabhakar et al., 2016). Open source cores that you can download for free and modify should also lower the cost of hardware design.

Pitfall *Performance counters added as an afterthought for DSA hardware.*

The TPU has 106 performance counters, and the designers wanted even more (see Figure 7.45). The *raison d'être* for *DSAs* is performance, and it is way too early in their evolution to have a good idea about what is going on.

Fallacy *Architects are tackling the right DNN tasks.*

The architecture community is paying attention to deep learning: 15% of the papers at ISCA 2016 were on hardware accelerators for DNNs! Alas, all nine papers looked at CNNs, and only two mentioned other DNNs. CNNs are more complex than MLPs and are prominent in DNN competitions (Russakovsky et al., 2015), which might explain their allure, but they are only about 5% of the Google data center NN workload. It seems wise try to accelerate MLPs and LSTMs with at least as much gusto.

Fallacy *For DNN hardware, inferences per second (IPS) is a fair summary performance metric.*

IPS is not appropriate as a single, overall performance summary for DNN hardware because it's simply the inverse of the complexity of the typical inference in the application (e.g., the number, size, and type of NN layers). For example, the

TPU runs the 4-layer MLP1 at 360,000 IPS but the 89-layer CNN1 at only 4700 IPS; thus TPU IPS varies by 75X! Therefore using IPS as the single-speed summary is much more misleading for NN accelerators than MIPS or FLOPS is for traditional processors, so IPS should be even more disparaged. To compare DNN machines better, we need a benchmark suite written at a high level to port it to the wide variety of DNN architectures. Fathom is a promising new attempt at such a benchmark suite (Adolf et al., 2016).

Pitfall *Being ignorant of architecture history when designing a DSA.*

Ideas that didn't fly for general-purpose computing may be ideal for DSAs, thus history-aware architects could have a competitive edge. For the TPU, three important architectural features date back to the early 1980s: systolic arrays (Kung and Leiserson, 1980), decoupled-access/execute (Smith, 1982b), and CISC instructions (Patterson and Ditzel, 1980). The first reduced the area and power of the large Matrix Multiply Unit, the second fetched weights concurrently during operation of the Matrix Multiply Unit, and the third better utilized the limited bandwidth of the PCIe bus for delivering instructions. We advise mining the historical perspectives sections at the end of every chapter of this book to discover jewels that could embellish DSAs that you design.

7.11 Concluding Remarks

In this chapter, we've seen several commercial examples of the recent shift from the traditional goal of improving general-purpose computers so that all programs benefit to accelerating a subset of programs with DSAs.

Both versions of Catapult preserved data-center homogeneity by designing a small, low-power FPGA board that could fit inside a server. The hope is that the flexibility of FPGAs will allow Catapult to be useful to many current applications and the new ones that appeared after deployment. Catapult runs search rank and CNNs faster than GPUs, offering a 1.5–1.75 gain in performance/TCO for ranking over CPUs.

The TPU project actually began with FPGAs but abandoned them when the designers concluded that the FPGAs of that time were not competitive in performance compared to the GPUs. They also believed the TPU would use much less power than GPUs, while being as fast or faster, potentially making the TPU much better than FPGAs and GPUs. Finally, the TPU was not the device that broke data center homogeneity at Google because some servers in its data centers already had GPUs. The TPU basically followed in the footsteps of the GPU and was just another type of accelerator.

The nonrecurring engineering costs were likely much higher for the TPU than for Catapult, but the rewards were also greater: both performance and performance/watt were much higher for an ASIC than for an FPGA. The risk was that the TPU was appropriate only for DNN inference, but as we mentioned, DNNs are an attractive target because they can potentially be used for many applications. In 2013

Google's management took a leap of faith by trusting that the DNN requirements in 2015 and beyond would justify investment in the TPU.

The deterministic execution model of both Catapult and the TPU is a better match to the response-time deadline of user-facing applications than are the time-varying optimizations of CPUs and GPUs (caches, out-of-order execution, multithreading, multiprocessing, prefetching, etc.) that help average throughput more than latency. The lack of such features helps explain why, despite having myriad ALUs and a big memory, the TPU is relatively small and low powered. This achievement suggests a "Cornucopia Corollary" to Amdahl's Law: *low utilization of a huge, cheap resource can still deliver high, cost-effective performance*.

In summary, the TPU succeeded for DNNs because of the large matrix unit; the substantial software-controlled on-chip memory; the ability to run whole inference models to reduce dependence on host CPU; a single-threaded, deterministic execution model that proved to be a good match to 99th-percentile response-time limits; enough flexibility to match the DNNs of 2017 as well as of 2013; the omission of general-purpose features that enabled a small and low-power die despite the larger datapath and memory; the use of 8-bit integers by the quantized applications; and the fact that applications were written using TensorFlow, which made it easy to port them to the DSA at high-performance rather than having to rewrite them in order for them to run well on the very different hardware.

Pixel Visual Core demonstrated the constraints of designing a DSA for a PMD in terms of die size and power. Unlike the TPU, it is a separate processor from the host that fetches its own instructions. Despite being aimed primarily at computer vision, Pixel Visual Core can run CNNs one to two orders of magnitude better in performance/watt than the K80 GPU and the Haswell CPU.

It's too early to render judgment on the Intel Crest, although its enthusiastic announcement by the Intel CEO signals a shift in the computing landscape.

An Architecture Renaissance

For at least the past decade, architecture researchers have been publishing innovations based on simulations using limited benchmarks claiming improvements for general-purpose processors of *10% or less* while companies are now reporting gains for DSA hardware products of *10 times or more*.

We think that is a sign that the field is undergoing a transformation, and we expect to see a renaissance in architecture innovation in the next decade because of

- the historic end of both Dennard scaling and Moore's Law, which means improving cost-energy-performance will require innovation in computer architecture;
- the productivity advances in building hardware from both Agile hardware development and new hardware design languages that leverage advances in modern programming languages;

- the reduced cost of hardware development because of free and open instruction sets, open-source IP blocks, and commercial IP blocks (which so far is where most DSAs are found);
- the improvements mentioned above in productivity and cost of development means researchers can afford to demonstrate their ideas by building them in FPGAs or even in custom chips, instead of trying to convince skeptics with simulators; and
- the potential upside of DSAs and their synergy with domain-specific programming languages.

We believe that many architecture researchers will build DSAs that will raise the bar still higher than those discussed in this chapter. And we can't wait to see what the computer architecture world will look like by the next edition of this book!

7.12 Historical Perspectives and References

Section M.9 (available online) covers the development of DSAs.

Case Studies and Exercises by Cliff Young

Case Study: Google's Tensor Processing Unit and Acceleration of Deep Neural Networks

Concepts illustrated by this case study

- Structure of matrix multiplication operations
- Capacities of memories and rates of computations ("speeds and feeds") for a simple neural network model
- Construction of a special-purpose ISA
- Inefficiencies in mapping convolutions to TPU hardware
- Fixed-point arithmetic
- Function approximation

7.1 [10/20/10/25/25] <7.3,7.4> Matrix multiplication is a key operation supported in hardware by the TPU. Before going into details of the TPU hardware, it's worth analyzing the matrix multiplication calculation itself. One common way to depict matrix multiplication is with the following triply nested loop:

```
float a[M][K], b[K][N], c[M][N];
// M, N, and K are constants.
for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
                for (int k = 0; k < K; ++k)
                        c[i][j] += a[i][k] * b[k][j];
```

a. [10] Suppose that M, N, and K are all equal. What is the asymptotic complexity in time of this algorithm? What is the asymptotic complexity in space of the arguments? What does this mean for the operational intensity of matrix multiplication as M, N, and K grow large?

b. [20] Suppose that M=3, N=4, and K=5, so that each of the dimensions are relatively prime. Write out the order of accesses to memory locations in each of the three matrices A, B, and C (you might start with two-dimensional indices, then translate those to memory addresses or offsets from the start of each matrix). For which matrices are the elements accessed sequentially? Which are not? Assume row-major (C-language) memory ordering.

c. [10] Suppose that you transpose matrix B, swapping its indices so that they are B[N][K] instead. So, now the innermost statement of the loop looks like:

```
c[i][j] += a[i][k] * b[j][k];
```

Now, for which matrices are the elements accessed sequentially?

d. [25] The innermost (k-indexed) loop of our original routine performs a dot-product operation. Suppose that you are a given a hardware unit that can perform an 8-element dot-product more efficiently than the raw C code, behaving effectively like this C function:

```
void hardware_dot(float *accumulator,
   const float *a_slice, const float *b_slice) {
       float total = 0.;
       for (int k = 0; k < 8; ++k) {
              total += a_slice[k] * b_slice[k];
       }
       *accumulator += total;
}
```

How would you rewrite the routine with the transposed B matrix from part (c) to use this function?

e. [25] Suppose that instead, you are given a hardware unit that performs an 8-element “saxpy” operation, which behaves like this C function:

```
void hardware_saxpy(float *accumulator,
   float a, const float *input) {
       for (int k = 0; k < 8; ++k) {
              accumulator[k] += a * input[k];
       }
}
```

Write another routine that uses the saxpy primitive to deliver equivalent results to the original loop, without the transposed memory ordering for the B matrix.

7.2 [15/10/10/20/15/15/20/20] <7.3,7.4> Consider the neural network model MLP0 from Figure 7.5. That model has 20 M weights in five fully connected layers (neural network researchers count the input layer as if it were a layer in the stack, but it has no weights associated with it). For simplicity, let's assume that those layers are each the same size, so each layer holds 4 M weights. Then assume that each layer has identical geometry, so each group of 4 M weights represents a 2 K*2 K matrix. Because the TPU typically uses 8-bit numerical values, 20 M weights take up 20 MB.

a. [15] For batch sizes of 128, 256, 512, 1024, and 2048, how big are the input activations for each layer of the model (which, except for the input layer, are also the output activations of the previous layer)? Now considering the whole mode (i.e., there's just the input to the first layer and the output from the last layer), for each batch size, what is the transfer time for input and output over PCIe Gen3 x16, which has a transfer speed of about 100 Gibit/s?

b. [10] Given the memory system speed of 30 GiB/s, give a lower bound for the time the TPU takes to read the weights of MLP0 from memory. How much time does it take for the TPU to read a 256 × 256 "tile" of weights from memory?

c. [10] Show how to calculate the TPU's 92 T operations/second, given that we know that the systolic array matrix multiplier has 256 × 256 elements, each of which performs an 8-bit multiply-accumulate operation (MAC) each cycle. In high-performance-computing marketing terms, a MAC counts as two operations.

d. [20] Once a weight tile has been loaded into the matrix unit of the TPU, it can be reused to multiply a 256-element input vector by the 256 × 256 weight matrix represented by the tile to produce a 256-element output vector every cycle. How many cycles pass during the time it takes to load a weight tile? This is the "break-even" batch size, where compute and memory-load times are equal, also known as the "ridge" of the roofline.

e. [15] The compute peak for the Intel Haswell x86 server is about 1 T FLOPS, while the compute peak for the NVIDIA K80 GPU is about 3 T FLOPS. Supposing that they hit these peak numbers, calculate their best-case compute time for batch size 128. How do these times compare to the time the TPU takes to load all 20 M weights from memory?

f. [15] Assuming that the TPU program does not overlap computation with I/O over PCIe, calculate the time elapsed from when the CPU starts to send the first byte of data to the TPU until the time that the last byte of output is returned. What fraction of PCIe bandwidth is used?

g. [20] Suppose that we deployed a configuration where one CPU was connected to five TPUs across a single PCIe Gen3 x16 bus (with appropriate PCIe switches). Assume that we parallelize by placing one layer of MLP0 on each TPU, and that the TPUs can communicate directly with each other over PCIe. At batch = 128, what is the best-case latency for calculating a single inference, and what throughput, in terms of inferences per second, would such a configuration deliver? How does this compare to a single TPU?

h. [20] Suppose that each example in a batch of inferences requires 50 core-microseconds of processing time on the CPU. How many cores on the host CPU will be required to drive a single-TPU configuration at batch = 128?

7.3 [20/25/25/25/Discussion] <7.3,7.4> Consider a pseudo-assembly language for the TPU, and consider the program that handles a batch of size 2048 for a tiny fully connected layer with a 256 × 256 weight matrix. If there were no constraints on the sizes or alignments of computations in each instruction, the entire program for that layer might look like the following:

```
read_host u#0, 256*2048
read_weights w#0, 256*256
// matmul weights are implicitly read from the FIFO.
activate u#256*2048, a#0, 256*2048
write_host, u#256*2048, 256*2048
```

In this pseudo-assembly language, a prefix of “u#” refers to a memory address in the unified buffer; a prefix of “w#” refers to a memory address in the off-chip weight DRAM, and a prefix of “a#” refers to an accumulator address. The last argument of each assembly instruction describes the number of bytes to be operated upon.

Let’s walk through the program instruction by instruction:

- The read_host instruction reads 512 KB of data from host memory, storing it at the very beginning of the unified buffer (u#0).
- The read_weights instruction tells the weight fetching unit to read 64 KB of weights, loading them into the on-chip weight FIFO. These 64 KB of weights represent a single, 256 × 256 matrix of weights, which we will call a “weight tile.”
- The matmul instruction reads the 512 KB of input data from address 0 in the unified buffer, performs a matrix multiplication with the tile of weights, and stores the resulting 256*2048 = 524,288, 32-bit activations at accumulator address 0 (a#0). We have intentionally glossed over the details of the ordering of weights; the exercise will expand on these details.
- The activate instruction takes those 524,288 32-bit accumulators at a#0, applies an activation function to them, and stores the resulting 524,288, 8-bit output values at the next free location in the unified buffer, u#524288.
- The write_host instruction writes the 512 KB of output activations, starting at u#524288, back to the host CPU.

We will progressively add realistic details to the pseudo-assembly language to explore some aspects of TPU design.

a. [20] While we have written our pseudo-code in terms of bytes and byte addresses (or in the case of the accumulators, in terms of addresses to 32-bit values), the TPU operates on a natural vector length of 256. This means that the unified buffer is typically addressed at 256-byte boundaries, the accumulators are addressed in groups of 256 32-bit values (or at 1 KB boundaries), and weights are loaded in groups of 65,536 8-bit values. Rewrite the program’s addresses and transfer sizes to take these vector and weight-tile lengths into account. How many 256-element vectors of input activations will be read by the program? How many bytes of accumulated values will be used while computing the results? How many 256-element vectors of output activations will be written back to the host?

b. [25] Suppose that the application requirements change, and instead of a multiplication by a 256 × 256 weight matrix, the shape of the weight matrix now becomes 1024 × 256. Think of the matmul instruction as putting the weights as the right argument of the matrix multiplication operator, so 1024 corresponds to K, the dimension in which the matrix multiplication adds up values. Suppose that there are now two variants of the accumulate instruction, one of which overwrites the accumulators with its results, and the other of which adds the matrix multiplication results to the specified accumulator. How would you change the program to handle this 1024 × 256 matrix? Do you need more accumulators? The size of the matrix unit remains the same at 256 × 256; how many 256 × 256 weight tiles does your program need?

c. [25] Now write the program to handle a multiplication by a weight matrix of size 256 × 512. Does your program need more accumulators? Can you rewrite your program so that it uses only 2048, 256-entry accumulators? How many weight tiles does your program need? In what order should they be stored in the weight DRAM?

d. [25] Next, write the program to handle a multiplication by a weight matrix of size 1024 × 768. How many weight tiles does your program need? Write your program so that it uses only 2048, 256-entry accumulators. In what order should the weight tiles be stored in the weight DRAM? For this calculation, how many times did each input activation get read?

e. [Discussion] What would it take to build an architecture that reads each 256-element set of input activations just once? How many accumulators would that require? If you did it that way, how big would the accumulator memory have to be? Contrast this approach with the TPU, which uses 4096 accumulators, so that one set of 2048 accumulators can be written by the matrix unit while another is being used for activations.

7.4 [15/15/15] <7.3,7.4> Consider the first convolutional layer of AlexNet, which uses a 7 × 7 convolutional kernel, with an input feature depth of 3 and an output feature depth of 48. The original image width is 220 × 220.

a. [15] Ignore the 7 × 7 convolutional kernel for the moment, and consider just the center element of that kernel. A 1 × 1 convolutional kernel is mathematically equivalent to a matrix multiplication, using a weight matrix that is input_depth × output_depth in dimensions. With these depths, and using a standard matrix multiplication, what fraction of the TPU's 65,536 ALUs can be used?

b. [15] For convolutional neural networks, the spatial dimensions are also sources of weight reuse, since the convolutional kernel gets applied to many different (x,y) coordinate positions. Suppose that the TPU reaches balanced compute and memory at a batch size of 1400 (as you might have computed in exercise 1d). What is the smallest square image size that the TPU can process efficiently at a batch size of 1?

c. [15] The first convolutional layer of AlexNet implements a *kernel stride* of 4, which means that rather than moving by one X or Y pixel at each application, the 7 × 7 kernel moves by 4 pixels at a time. This striding means that we can permute the input data from 220 × 220 × 3 to be 55 × 55 × 48 (dividing the X and Y dimensions by 4 and multiplying the input depth by 16), and simultaneously we can restack the 7 × 7 × 3 × 48 convolutional weights to be 2 × 2 × 48 × 48 (just as the input data gets restacked by 4 in X and Y, we do the same to the 7 × 7 elements of the convolutional kernel, ending up with ceiling(7/4) = 2 elements in each of the X and Y dimensions). Because the kernel is now 2 × 2, we need to perform only four matrix multiplication operations, using weight matrices of size 48 × 48. What is the fraction of the 65,536 ALUs that can be used now?

7.5 [15/10/20/20/20/25] < 7.3 > The TPU uses *fixed-point arithmetic* (sometimes also called *quantized arithmetic*, with overlapping and conflicting definitions), where integers are used to represent values on the real number line. There are a number of different schemes for fixed-point arithmetic, but they share the common theme that there is an affine projection from the integer used by hardware to the real number that the integer represents. An affine projection has the form $r = i^*s + b$, where i is the integer, r is the represented real value, and s and b are a scale and bias. You can of course write the projection in either direction, from integers to reals or vice versa (although you need to round when converting from reals to integers).

a. [15] The simplest activation function supported by the TPU is "ReLUX," which is a rectified linear unit with a maximum of X. For example, ReLU6 is defined by Relu6(x) = { 0, when $x < 0$; x, when $0 <= x <= 6$; and 6, when $x > 6$ }. So 0.0 and 6.0 on the real number line are the minimum and maximum values that Relu6 might produce. Assume that you use an 8-bit unsigned integer in hardware, and that you want to make 0 map to 0.0 and 255 map to 6.0. Solve for s and b.

b. [10] How many values on the real number line are exactly representable by an 8-bit quantized representation of ReLU6 output? What is the real-number spacing between them?

c. [20] The difference between representable values is sometimes called a "unit in the least place," or *ulp*, when performing numerical analysis. If you map a real number to its fixed-point representation, then map back, you only rarely get back the original real number. The difference between the original number and its representation is called the *quantization error*. When mapping a real number in the range [0.0,6.0] to an 8-bit integer, show that the worst-case quantization error is one-half of an ulp (make sure you round to the nearest representable value). You might consider graphing the errors as a function of the original real number.

d. [20] Keep the real-number range [0.0,6.0] for an 8-bit integer from the last step. What 8-bit unsigned integer represents 1.0? What is the quantization error for 1.0? Suppose that you ask the TPU to add 1.0 to 1.0. What answer do you get back, and what is the error in that result?

e. [20] If you pick a random number uniformly in the range [0.0, 6.0], then quantize it to an 8-bit unsigned integer, what distribution would you expect to see for the 256 integer values?

f. [25] The hyperbolic tangent function, *tanh*, is another commonly used activation function in deep learning: $\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$

Tanh also has a bounded range, mapping the entire real number line to the interval $(-1.0, 1.0)$. Solve for s and b for this range, using an 8-bit unsigned representation. Then solve for s and b using an 8-bit two's complement representation. For both cases, what real number does the integer 0 represent? Which integer represents the real number 0.0? Can you imagine any issues that might result from the quantization error incurred when representing 0.0?

7.6 [20/25/15/15/30/30/30/40/40/25/20/Discussion] <7.3> In addition to tanh, another s-shaped smooth function, the logistic sigmoid function $y = 1/(1+\exp(-x))$,

$$logistic\_sigmoid(x) = \frac{1}{1+e^{-x}}$$

is commonly used as an activation function in neural networks. A common way to implement them in fixed-point arithmetic uses a piecewise quadratic approximation, where the most significant bits of the input value select which table entry to use. Then the least significant bits of the input value are sent to a degree-2 polynomial that describes a parabola that is fit to the subrange of the approximated function.

a. [20] Using a graphing tool (we like www.desmos.com/calculator), draw the graphs for the logistic sigmoid and tanh functions.

b. [25] Now draw the graph of $y = \tanh(x/2)/2$. Compare that graph with the logistic sigmoid function. How much do they differ by? Build an equation that shows how to transform one into the other. Prove that your equation is correct.

c. [15] Given this algebraic identity, do you need to use two different sets of coefficients to approximate logistic sigmoid and tanh?

d. [15] Tanh is an odd function, meaning that $f(-x) = -f(x)$. Can you exploit this fact to save table space?

e. [30] Let's focus our attention on approximating *tanh* over the interval $x \in [0.0, 6.4]$ on the number line. Using floating-point arithmetic, write a program that divides the interval into 64 subintervals (each of length 0.1), and then approximates the value of *tanh* over each subinterval using a single constant floating-point value (so you'll need to pick 64 different floating-point values, one for each subinterval). If you spot-check 100 different values (randomly chosen is fine) within each subinterval, what is the worst-case approximation error you see over all subintervals? Can you choose your constant to minimize the approximation error for each subinterval?

f. [30] Now consider building a floating-point linear approximation for each subinterval. In this case, you want to pick a pair of floating-point values m and b, for the traditional line equation $y = mx + b$, to approximate each of the 64 subintervals. Come up with a strategy that you think is reasonable to build this linear interpolation over 64 subintervals for tanh. Measure the worst-case approximation error over the 64 intervals. Is your approximation monotonic when it reaches a boundary between subintervals?

g. [40] Next, build a quadratic approximation, using the standard formula $y = ax^2 + bx + c$. Experiment with a number of different ways to fit the formula. Try fitting the parabola to the endpoints and midpoint of the bucket, or using a Taylor approximation around a single point in the bucket. What worst-case error do you get?

h. [40] (extra credit) Let's combine the numerical approximations of this exercise with the fixed-point arithmetic of the previous exercise. Suppose that the input $x \in [0.0, 6.4]$ is represented by a 15-bit unsigned value, with 0x0000 representing 0.0 and 0x7FFF representing 6.4. For the output, similarly use a 15-bit unsigned value, with 0x0000 representing 0.0 and 0x7FFF representing 1.0. For each of your constant, linear, and quadratic approximations, calculate the combined effect of approximation and quantization errors. Since there are so few input values, you can write a program to check them exhaustively.

i. [25] For the quadratic, quantized approximation, is your approximation monotonic within each subinterval?

j. [20] A difference of one ulp in the output scale should correspond to an error of 1.0 / 32767. How many ulps of error are you seeing in each case?

k. [Discussion] By choosing to approximate the interval [0.0, 6.4], we effectively clipped the "tail" of the hyperbolic tangent function, for values of $x > 6.4$. It's not an unreasonable approximation to set the output value for all of the tail to 1.0. What's the worst-case error, in terms of both real numbers and ulps, of treating the tail this way? Is there a better place we might have clipped the tail to improve our accuracy?

Exercises

7.7 [10/20/10/15] <7.2,7.5> One popular family of FPGAs, the Virtex-7 series, is built by Xilinx. A Virtex-7 XC7VX690T FPGA contains 3,600 25x18-bit integer multiply-add "DSP slices." Consider building a TPU-style design on such an FPGA.

a. [10] Using one 25 × 18 integer multiplier per systolic array cell, what's the largest matrix multiplication unit one could construct? Assume that the matrix multiplication unit must be square.

b. [20] Suppose that you could build a rectangular, nonsquare matrix multiplication unit. What implications would such a design have for hardware and software? (Hint: think about the vector length that software must handle.)

c. [10] Many FPGA designs are lucky to reach 500 MHz operation. At that speed, calculate the peak 8-bit operations per second that such a device might achieve. How does that compare to the 3 T FLOPS of a K80 GPU?

d. [15] Assume that you can make up the difference between 3600 and 4096 DSP slices using LUTs, but that doing so will reduce your clock rate to 350 MHz. Is this a worthwhile trade-off to make?

7.8 [15/15/15] <7.9> Amazon Web Services (AWS) offers a wide variety of "computing instances," which are machines configured to target different applications and scales. AWS prices tell us useful data about the Total Cost of Ownership (TCO) of various computing devices, particularly as computer equipment is often depreciated[1] on a 3-year schedule. As of July 2017, a dedicated, compute-oriented "c4" computing instance includes two x86 chips with 20 physical cores in total. It rents on-demand for $1.75/hour, or $17,962 for 3 years. In contrast, a dedicated "p2" computing instance also has two x86 chips but with 36 cores in total, and adds 16 NVIDIA K80 GPUs. A p2 rents on-demand for $15.84/hour, or $184,780 for 3 years.

a. [15] The c4 instance uses Intel Xeon E5-2666 v3 (Haswell) processors. The p2 instance uses Intel Xeon E5-2686 v4 (Broadwell) processors. Neither part number is listed officially on Intel's product website, which suggests that these parts are specially built for Amazon by Intel. The E5-2660 v3 part has a similar core count to the E5-2666 v3 and has a street price of around $1500. The E5-2697 v4 part has a similar core count to the E5-2686 v4 and has a street price of around $3000. Assume that the non-GPU portion of the p2 instance would have a price proportional to the ratio of street prices. What is the TCO, over 3 years, for a single K80 GPU?

b. [15] Suppose that you have a compute- and throughput-dominated workload that runs at rate 1 on the c4 instance and at rate T on the GPU-accelerated p2 instance. How large must T be for the GPU-based solution to be more cost-effective? Suppose that each general-purpose CPU core can compute at a rate of about 30G single-precision FLOPS. Ignoring the CPUs of the p2 instance, what fraction of peak K80 FLOPs would be required to reach the same rate of computation as the c4 instance?

c. [15] AWS also offers "f1" instances that include 8 Xilinx Ultrascale + VU9P FPGAs. They rent at $13.20/hour, or $165,758 for 3 years. Each VU9P device includes 6840 DSP slices, which can perform 27×18-bit integer multiply-accumulate operations (recall that one multiply-accumulate counts as two "operations"). At 500 MHz, what is the peak multiply-accumulate operations/cycle that an f1-based system might achieve, counting all 8 FPGAs toward the computation total? Assuming that the integer operations on the FPGAs can substitute for floating-point operations, how does this compare to the peak single-precision multiply-accumulate operations/cycle of the GPUs of the p2 instance? How do they compare in terms of cost-effectiveness?

7.9 [20/20/25] <7.7> As shown in Figure 7.34 (but simplified to fewer PEs), each Pixel Visual Core includes a 16×16 set of full processing elements, surrounded by an additional two layers of "simplified" processing elements. Simplified PEs can store and communicate data but omit the computation hardware of full PEs. Simplified PEs store copies of data that might be the "home data" of a neighboring core, so there are $(16+2+2)^2=400$ PEs in total, 256 full and 144 simplified.

---

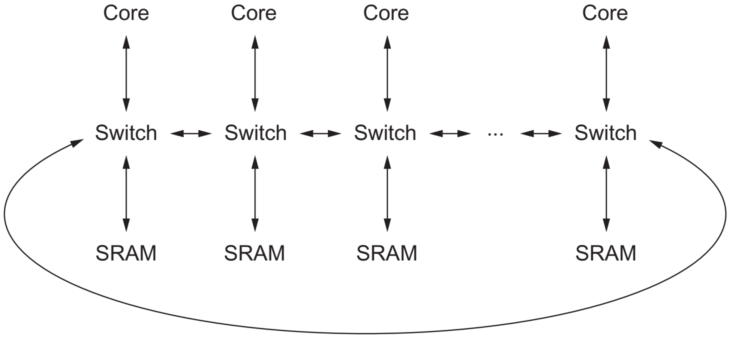
[1] Capital expenses are accounted for over the lifetime of an asset, using a "depreciation schedule." Rather than taking a one-time charge at the point where an asset is acquired, standard accounting practice spreads out the capital cost over the lifetime of the asset. So one might account for a $30,000 device that has a useful life of 3 years by assigning $10,000 in depreciation to each year.

a. [20] Suppose that you wanted to process a 64 × 32 grayscale image with a 5 × 5 stencil using 8 Pixel Visual Cores. For now, assume that the image is laid out in raster-scan order (pixels that are adjacent in X are adjacent in memory, while pixels that are adjacent in Y are 64 memory locations apart). For each of the 8 cores, describe the memory region that the core should import to handle its part of the image. Make sure to include the halo region. Which parts of the halo region should be zeroed by software to ensure correct operation? You may find it convenient to refer to subregions of the image using a 2D slice notation, where for example image[2:5][6:13] refers to the set of pixels whose x component is $2<=x<5$ and whose y component is $6<=y<13$ (the slices are half-open following Python slicing practice).

b. [20] If we change to a 3 × 3 stencil, how do the regions imported from memory change? How many halo-simplified PEs go unused?

c. [25] Now consider how to support a 7 × 7 stencil. In this case, we don't have as many hardware-supported simplified PEs as we need to cover the three pixels worth of halo data that "belong to" neighboring cores. To handle this, we use the outermost ring of full PEs as if they were simplified PEs. How many pixels can we handle in a single core using this strategy? How many "tiles" are now required to handle our 64 × 32 input image? What is the utilization of our full PEs over the complete processing time for the 7 × 7 stencil over the 64 × 32 image?

7.10 [20/20/20/25/25] <7.7> Consider a case in which each of the eight cores on a Pixel Visual Core device is connected through a four-port switch to a 2D SRAM, forming a core + memory unit. The remaining two ports on the switch link these units in a ring, so that each core is able to access any of the eight SRAMs. However, this ring-based network-on-chip topology makes some data access patterns more efficient than others.



a. [20] Suppose that each link in the NOC has the same bandwidth B, and that each link is full-duplex, so it can simultaneously transfer bandwidth B in each direction. Links connect the core to the switch, the switch to SRAM, and pairs of switches in the ring. Assume that each local memory has at least B bandwidth, so it can saturate its link. Consider a memory access pattern where each of the eight PEs access only the closest memory (the one connected via the switch of the core + memory unit). What is the maximum memory bandwidth that the core will be able to achieve?

b. [20] Now consider an off-by-one access pattern, where core i accesses memory $i+1$, going through three links to reach that memory (core 7 will access memory 0, because of the ring topology). What is the maximum memory bandwidth that the core will be able to achieve in this case? To achieve that bandwidth, do you need to make any assumptions about the capabilities of the 4-port switch? What if the switch can only move data at rate B?

c. [20] Consider an off-by-two access pattern, where core i access memory $i+2$. Once again, what is the maximum memory bandwidth that the core will be able to achieve in this case? Where are the bottleneck links in the network-on-chip?

d. [25] Consider a uniform random memory access pattern, where each core uses each of the SRAMs for ⅛ of its memory requests. Assuming this traffic pattern, how much traffic traverses a switch-to-switch link, compared to the amount of traffic between a core and its associated switch or between an SRAM and its associated switch?

e. [25] (advanced) Can you conceive of a case (workload) where this network can deadlock? From the standpoint of software-only solutions, what should the compiler do to avoid such a scenario? If you can make changes to hardware, what changes in routing topology (and routing scheme) would guarantee no deadlocks?

7.11 <7.2> The first Anton molecular dynamics supercomputer typically simulated a box of water that was 64 Å on a side. The computer itself might be approximated as a box with 1 m side length. A single simulation step represented 2.5 fs of simulation time, and took about 10 μs of wall-clock time. The physics models used in molecular dynamics act as if every particle in the system exerts a force on every other particle in the system on each ("outer") time step, requiring what amounts to a global synchronization across the entire computer.

a. Calculate the spatial expansion factor from simulation space to hardware in real space.

b. Calculate the temporal slowdown factor from simulated time to wall-clock time.

c. These two numbers come out surprisingly close. Is this just a coincidence, or is there some other limit that constrains them in some way? (Hint: the speed of light applies to both the simulated chemical system and the hardware that does the simulation.)

d. Given these limits, what would it take to use a warehouse-scale supercomputer to perform molecular dynamics simulations at Anton rates? That is, what's the fastest simulation step time that might be achieved with a machine 10^2 or 10^3 m on a side? What about simulating on a world-spanning Cloud service?

7.12 <7.2> The Anton communication network is a 3D, $8 \times 8 \times 8$ torus, where each node in the system has six links to neighboring nodes. Latency for a packet to transit single link is about 50 ns. Ignore on-chip switching time between links for this exercise.

a. What is the diameter (maximum number of hops between a pair of nodes) of the communication network? Given that diameter, what is the shortest latency required to broadcast a single value from one node of the machine to all 512 nodes of the machine?

b. Assuming that adding up two values takes zero time, what is the shortest latency to add up a sum over 512 values to a single node, where each value starts on a different node of the machine?

c. Once again assume that you want to perform the sum over 512 values, but you want each of the 512 nodes of the system to end up with a copy of the sum. Of course you could perform a global reduction followed by a broadcast. Can you do the combined operation in less time? This pattern is called an *all-reduce*. Compare the times of your all-reduce pattern to the time of a broadcast from a single node or a global sum to a single node. Compare the bandwidth used by the all-reduce pattern with the other patterns.

A.1 Introduction A-2
A.2 Classifying Instruction Set Architectures A-3
A.3 Memory Addressing A-7
A.4 Type and Size of Operands A-13
A.5 Operations in the Instruction Set A-15
A.6 Instructions for Control Flow A-16
A.7 Encoding an Instruction Set A-21
A.8 Cross-Cutting Issues: The Role of Compilers A-24
A.9 Putting It All Together: The RISC-V Architecture A-33
A.10 Fallacies and Pitfalls A-42
A.11 Concluding Remarks A-46
A.12 Historical Perspective and References A-47
Exercises by Gregory D. Peterson A-47

A

Instruction Set Principles

A *n*	Add the number in storage location *n* into the accumulator.
E *n*	If the number in the accumulator is greater than or equal to zero execute next the order which stands in storage location *n*; otherwise proceed serially.
Z	Stop the machine and ring the warning bell.

Wilkes and Renwick,
Selection from the List of 18 Machine
Instructions for the EDSAC (1949)

A.1 Introduction

In this appendix we concentrate on instruction set architecture—the portion of the computer visible to the programmer or compiler writer. Most of this material should be review for readers of this book; we include it here for background. This appendix introduces the wide variety of design alternatives available to the instruction set architect. In particular, we focus on four topics. First, we present a taxonomy of instruction set alternatives and give some qualitative assessment of the advantages and disadvantages of various approaches. Second, we present and analyze some instruction set measurements that are largely independent of a specific instruction set. Third, we address the issue of languages and compilers and their bearing on instruction set architecture. Finally, the "Putting It All Together" section shows how these ideas are reflected in the RISC-V instruction set, which is typical of RISC architectures. We conclude with fallacies and pitfalls of instruction set design.

To illustrate the principles further and to provide a comparison with RISC-V, Appendix K also gives four examples of other general-purpose RISC architectures (MIPS, Power ISA, SPARC, and Armv8), four embedded RISC processors (ARM Thumb2, RISC-V Compressed, microMIPS), and three older architectures (80x86, IBM 360/370, and VAX). Before we discuss how to classify architectures, we need to say something about instruction set measurement.

Throughout this appendix, we examine a wide variety of architectural measurements. Clearly, these measurements depend on the programs measured and on the compilers used in making the measurements. The results should not be interpreted as absolute, and you might see different data if you did the measurement with a different compiler or a different set of programs. We believe that the measurements in this appendix are reasonably indicative of a class of typical applications. Many of the measurements are presented using a small set of benchmarks, so that the data can be reasonably displayed and the differences among programs can be seen. An architect for a new computer would want to analyze a much larger collection of programs before making architectural decisions. The measurements shown are usually *dynamic*—that is, the frequency of a measured event is weighed by the number of times that event occurs during execution of the measured program.

Before starting with the general principles, let's review the three application areas from Chapter 1. *Desktop computing* emphasizes the performance of programs with integer and floating-point data types, with little regard for program size. For example, code size has never been reported in the five generations of SPEC benchmarks. *Servers* today are used primarily for database, file server, and Web applications, plus some time-sharing applications for many users. Hence, floating-point performance is much less important for performance than integers and character strings, yet virtually every server processor still includes floating-point instructions. *Personal mobile devices* and *embedded applications* value cost and energy, so code size is important because less memory is both cheaper and lower energy, and some classes of instructions (such as floating point) may be optional to reduce chip costs, and a compressed version of the instructions set designed to save memory space may be used.

Thus, instruction sets for all three applications are very similar. In fact, architectures similar to RISC-V, which we focus on here, have been used successfully in desktops, servers, and embedded applications.

One successful architecture very different from RISC is the 80x86 (see Appendix K). Surprisingly, its success does not necessarily belie the advantages of a RISC instruction set. The commercial importance of binary compatibility with PC software combined with the abundance of transistors provided by Moore's Law led Intel to use a RISC instruction set internally while supporting an 80x86 instruction set externally. Recent 80x86 microprocessors, including all the Intel Core microprocessors built in the past decade, use hardware to translate from 80x86 instructions to RISC-like instructions and then execute the translated operations inside the chip. They maintain the illusion of 80x86 architecture to the programmer while allowing the computer designer to implement a RISC-style processor for performance. There remain, however, serious disadvantages for a complex instruction set like the 80x86, and we discuss these further in the conclusions.

Now that the background is set, we begin by exploring how instruction set architectures can be classified.

A.2 Classifying Instruction Set Architectures

The type of internal storage in a processor is the most basic differentiation, so in this section we will focus on the alternatives for this portion of the architecture. The major choices are a stack, an accumulator, or a set of registers. Operands may be named explicitly or implicitly: The operands in a *stack architecture* are implicitly on the top of the stack, and in an *accumulator architecture* one operand is implicitly the accumulator. The *general-purpose register architectures* have only explicit operands—either registers or memory locations. Figure A.1 shows a block diagram of such architectures, and Figure A.2 shows how the code sequence `C = A + B` would typically appear in these three classes of instruction sets. The explicit operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of architecture and choice of specific instruction.

As the figures show, there are really two classes of register computers. One class can access memory as part of any instruction, called *register-memory* architecture, and the other can access memory only with load and store instructions, called *load-store* architecture. A third class, not found in computers shipping today, keeps all operands in memory and is called a *memory-memory* architecture. Some instruction set architectures have more registers than a single accumulator but place restrictions on uses of these special registers. Such an architecture is sometimes called an *extended accumulator* or *special-purpose register* computer.

Although most early computers used stack or accumulator-style architectures, virtually every new architecture designed after 1980 uses a load-store register architecture. The major reasons for the emergence of general-purpose register (GPR) computers are twofold. First, registers—like other forms of storage internal to the processor—are faster than memory. Second, registers are more efficient for a

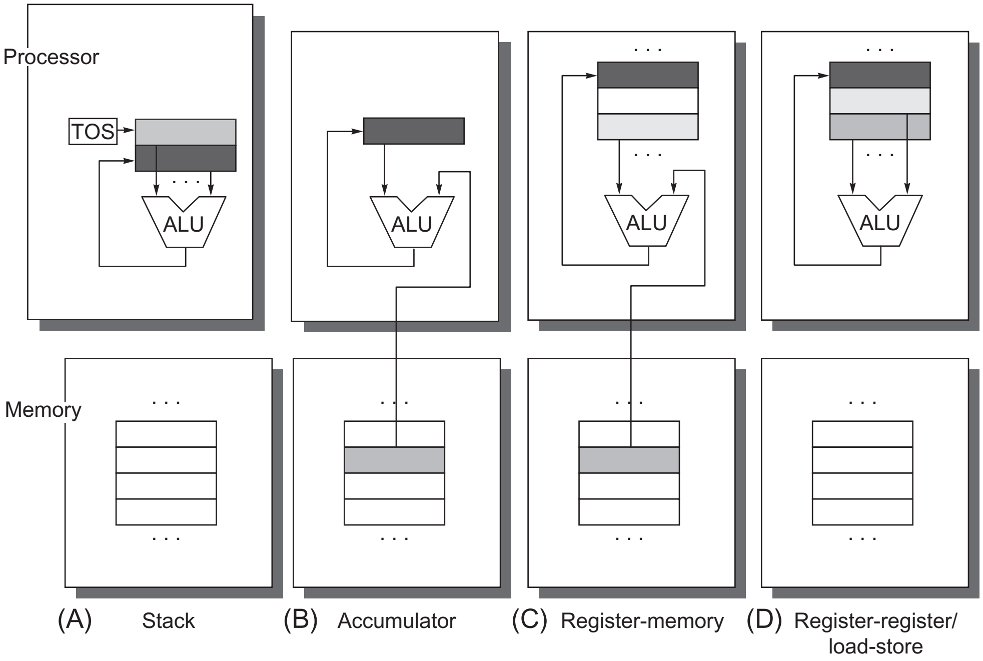


Figure A.1 Operand locations for four instruction set architecture classes. The arrows indicate whether the operand is an input or the result of the arithmetic-logical unit (ALU) operation, or both an input and result. Lighter shades indicate inputs, and the dark shade indicates the result. In (A), a top of stack (TOS) register points to the top input operand, which is combined with the operand below. The first operand is removed from the stack, the result takes the place of the second operand, and TOS is updated to point to the result. All operands are implicit. In (B), the accumulator is both an implicit input operand and a result. In (C), one input operand is a register, one is in memory, and the result goes to a register. All operands are registers in (D) and, like the stack architecture, can be transferred to memory only via separate instructions: push or pop for (A) and load or store for (D).

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R3,R1,B`	`Load R2,B`
`Add`	`Store C`	`Store R3,C`	`Add R3,R1,R2`
`Pop C`			`Store R3,C`

Figure A.2 The code sequence for `C = A + B` for four classes of instruction sets. Note that the `Add` instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the `Add` operation for each class of architecture.

compiler to use than other forms of internal storage. For example, on a register computer the expression `(A * B)+(B * C) - (A * D)` may be evaluated by doing the multiplications in any order, which may be more efficient because of the location of the operands or because of pipelining concerns (see Chapter 3). Nevertheless, on a stack computer the hardware must evaluate the expression in only one order, because operands are hidden on the stack, and it may have to load an operand multiple times.

More importantly, registers can be used to hold variables. When variables are allocated to registers, the memory traffic reduces, the program speeds up (because registers are faster than memory), and the code density improves (because a register can be named with fewer bits than can a memory location).

As explained in Section A.8, compiler writers would prefer that all registers be equivalent and unreserved. Older computers compromise this desire by dedicating registers to special uses, effectively decreasing the number of general-purpose registers. If the number of truly general-purpose registers is too small, trying to allocate variables to registers will not be profitable. Instead, the compiler will reserve all the uncommitted registers for use in expression evaluation.

How many registers are sufficient? The answer, of course, depends on the effectiveness of the compiler. Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables. Modern compiler technology and its ability to effectively use larger numbers of registers has led to an increase in register counts in more recent architectures.

Two major instruction set characteristics divide GPR architectures. Both characteristics concern the nature of operands for a typical arithmetic or logical instruction (ALU instruction). The first concerns whether an ALU instruction has two or three operands. In the three-operand format, the instruction contains one result operand and two source operands. In the two-operand format, one of the operands is both a source and a result for the operation. The second distinction among GPR architectures concerns how many of the operands may be memory addresses in ALU instructions. The number of memory operands supported by a typical ALU instruction may vary from none to three. Figure A.3 shows combinations of these two attributes with examples of computers. Although there are seven

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Figure A.3 Typical combinations of memory operands and total operands per typical ALU instruction with examples of computers. Computers with no memory reference per ALU instruction are called load-store or register-register computers. Instructions with multiple memory operands per typical ALU instruction are called register-memory or memory-memory, according to whether they have one or more than one memory operand.

Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see Appendix C)	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density	Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

Figure A.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (m, n) means m memory operands and n total operands. In general, computers with fewer alternatives simplify the compiler's task because there are fewer decisions for the compiler to make (see Section A.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size because you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes three extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

possible combinations, three serve to classify nearly all existing computers. As we mentioned earlier, these three are load-store (also called register-register), register-memory, and memory-memory.

Figure A.4 shows the advantages and disadvantages of each of these alternatives. Of course, these advantages and disadvantages are not absolutes: they are qualitative and their actual impact depends on the compiler and implementation strategy. A GPR computer with memory-memory operations could easily be ignored by the compiler and used as a load-store computer. One of the most pervasive architectural impacts is on instruction encoding and the number of instructions needed to perform a task. We see the impact of these architectural alternatives on implementation approaches in Appendix C and Chapter 3.

Summary: Classifying Instruction Set Architectures

Here and at the end of Sections A.3–A.8 we summarize those characteristics we would expect to find in a new instruction set architecture, building the foundation for the RISC-V architecture introduced in Section A.9. From this section we should clearly expect the use of general-purpose registers. Figure A.4, combined with Appendix C on pipelining, leads to the expectation of a load-store version of a general-purpose register architecture.

With the class of architecture covered, the next topic is addressing operands.

A.3

Memory Addressing

Independent of whether the architecture is load-store or allows any operand to be a memory reference, it must define how memory addresses are interpreted and how they are specified. The measurements presented here are largely, but not completely, computer independent. In some cases the measurements are significantly affected by the compiler technology. These measurements have been made using an optimizing compiler, because compiler technology plays a critical role.

Interpreting Memory Addresses

How is a memory address interpreted? That is, what object is accessed as a function of the address and the length? All the instruction sets discussed in this book are byte addressed and provide access for bytes (8 bits), half words (16 bits), and words (32 bits). Most of the computers also provide access for double words (64 bits).

There are two different conventions for ordering the bytes within a larger object. *Little Endian* byte order puts the byte whose address is “x … x000” at the least-significant position in the double word (the little end). The bytes are numbered:

7	6	5	4	3	2	1	0

Big Endian byte order puts the byte whose address is “x … x000” at the most-significant position in the double word (the big end). The bytes are numbered:

0	1	2	3	4	5	6	7

When operating within one computer, the byte order is often unnoticeable—only programs that access the same locations as both, say, words and bytes, can notice the difference. Byte order is a problem when exchanging data among computers with different orderings, however. Little Endian ordering also fails to match the normal ordering of words when strings are compared. Strings appear “SDRAWKCAB” (backwards) in the registers.

A second memory issue is that in many computers, accesses to objects larger than a byte must be *aligned*. An access to an object of size s bytes at byte address A is aligned if $A \bmod s = 0$. Figure A.5 shows the addresses at which an access is aligned or misaligned.

Why would someone design a computer with alignment restrictions? Misalignment causes hardware complications, because the memory is typically aligned on a multiple of a word or double-word boundary. A misaligned memory access may, therefore, take multiple aligned memory references. Thus, even in computers that allow misaligned access, programs with aligned accesses run faster.

	Value of three low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order three bits of the address.

Even if data are aligned, supporting byte, half-word, and word accesses requires an alignment network to align bytes, half words, and words in 64-bit registers. For example, in Figure A.5, suppose we read a byte from an address with its 3 low-order bits having the value 4. We will need to shift right 3 bytes to align the byte to the proper place in a 64-bit register. Depending on the instruction, the computer may also need to sign-extend the quantity. Stores are easy: only the addressed bytes in memory may be altered. On some computers a byte, half-word, and word operation does not affect the upper portion of a register. Although all the computers discussed in this book permit byte, half-word, and word accesses to memory, only the IBM 360/370, Intel 80x86, and VAX support ALU operations on register operands narrower than the full width.

Now that we have discussed alternative interpretations of memory addresses, we can discuss the ways addresses are specified by instructions, called *addressing modes*.

Addressing Modes

Given an address, we now know what bytes to access in memory. In this subsection we will look at addressing modes—how architectures specify the address of an object they will access. Addressing modes specify constants and registers in addition to locations in memory. When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.

Figure A.6 shows all the data addressing modes that have been used in recent computers. Immediates or literals are usually considered memory addressing modes (even though the value they access is in the instruction stream), although registers are often separated because they don't usually have memory addresses. We have kept addressing modes that depend on the program counter, called *PC-relative addressing*, separate. PC-relative addressing is used primarily for specifying code addresses in control transfer instructions, discussed in Section A.6.

Addressing mode	**Example instruction**	**Meaning**	**When used**
Register	`Add R4,R3`	`Regs[R4]←Regs[R4]+Regs[R3]`	When a value is in a register
Immediate	`Add R4,3`	`Regs[R4]←Regs[R4]+3`	For constants
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4]+Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4]+Mem[Regs[R1]]`	Accessing using a pointer or a computed address
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←Regs[R3]+Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: `R1`=base of array; `R2`=index amount
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1]+Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1]+Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer *p*, then mode yields **p*
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1]+Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+`*d*	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, *d*
Autodecrement	`Add R1, -(R2)`	`Regs[R2]←Regs[R2] -` *d* `Regs[R1]←Regs[R1]+Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1]+Mem[100+Regs[R2]+Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A.38.

Figure A.6 shows the most common names for the addressing modes, though the names differ among architectures. In this figure and throughout the book, we will use an extension of the C programming language as a hardware description notation. In this figure, only one non-C feature is used: the left arrow (←) is used for assignment. We also use the array `Mem` as the name for main memory and the array `Regs` for registers. Thus, `Mem[Regs[R1]]` refers to the contents of the memory location whose address is given by the contents of register 1 (`R1`). Later, we will introduce extensions for accessing and transferring data smaller than a word.

Addressing modes have the ability to significantly reduce instruction counts; they also add to the complexity of building a computer and may increase the average clock cycles per instruction (CPI) of computers that implement those modes. Thus, the usage of various addressing modes is quite important in helping the architect choose what to include.

Figure A.7 shows the results of measuring addressing mode usage patterns in three programs on the VAX architecture. We use the old VAX architecture for a few measurements in this appendix because it has the richest set of addressing

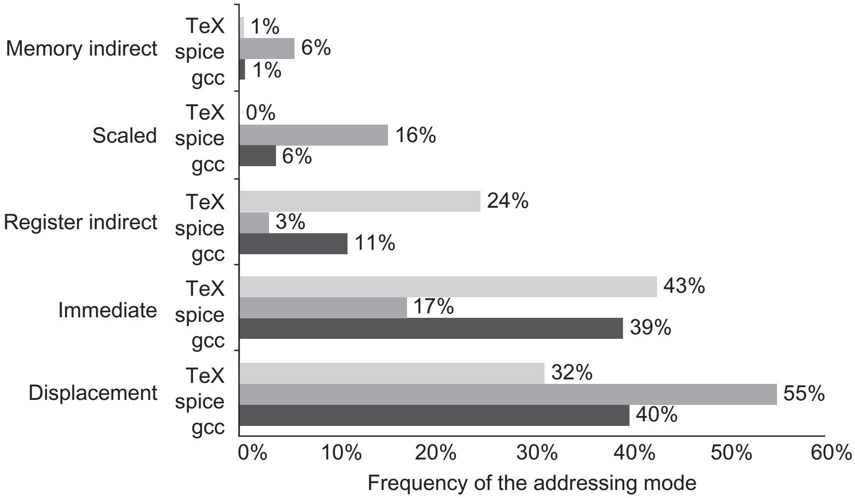


Figure A.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0%–3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost all the memory indirect references use displacement mode as the base. Displacement mode includes all displacement lengths (8, 16, and 32 bits). The PC-relative addressing modes, used almost exclusively for branches, are not included. Only the addressing modes with an average frequency of over 1% are shown.

modes and the fewest restrictions on memory addressing. For example, Figure A.6 on page A.9 shows all the modes the VAX supports. Most measurements in this appendix, however, will use the more recent register-register architectures to show how programs use instruction sets of current computers.

As Figure A.7 shows, displacement and immediate addressing dominate addressing mode usage. Let's look at some properties of these two heavily used modes.

Displacement Addressing Mode

The major question that arises for a displacement-style addressing mode is that of the range of displacements used. Based on the use of various displacement sizes, a decision of what sizes to support can be made. Choosing the displacement field sizes is important because they directly affect the instruction length. Figure A.8

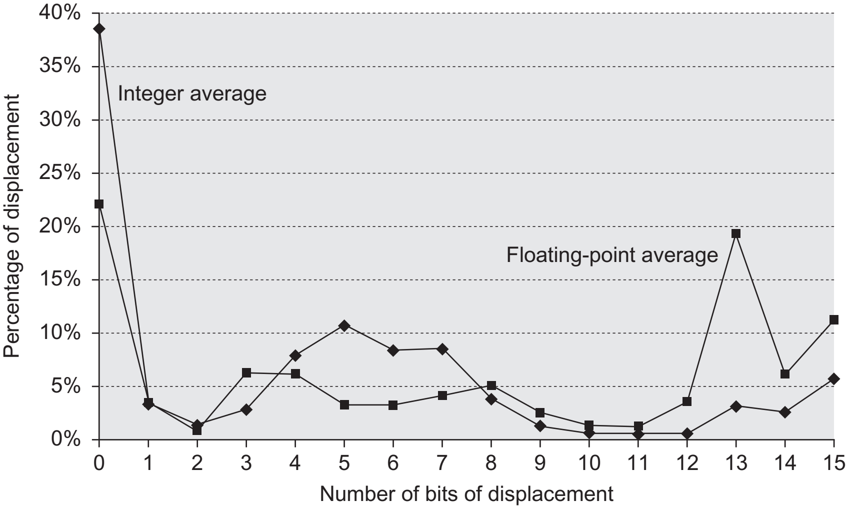


Figure A.8 Displacement values are widely distributed. There are both a large number of small values and a fair number of large values. The wide distribution of displacement values is due to multiple storage areas for variables and different displacements to access them (see Section A.8) as well as the overall addressing scheme the compiler uses. The *x*-axis is $\log_2$ of the displacement, that is, the size of a field needed to represent the magnitude of the displacement. Zero on the *x*-axis shows the percentage of displacements of value 0. The graph does not include the sign bit, which is heavily affected by the storage layout. Most displacements are positive, but a majority of the largest displacements (14+ bits) are negative. Because these data were collected on a computer with 16-bit displacements, they cannot tell us about longer displacements. These data were taken on the Alpha architecture with full optimization (see Section A.8) for SPEC CPU2000, showing the average of integer programs (CINT2000) and the average of floating-point programs (CFP2000).

shows the measurements taken on the data access on a load-store architecture using our benchmark programs. We look at branch offsets in Section A.6—data accessing patterns and branches are different; little is gained by combining them, although in practice the immediate sizes are made the same for simplicity.

Immediate or Literal Addressing Mode

Immediates can be used in arithmetic operations, in comparisons (primarily for branches), and in moves where a constant is wanted in a register. The last case occurs for constants written in the code—which tend to be small—and for address constants, which tend to be large. For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset. Figure A.9 shows the frequency of immediates for the general classes of integer and floating-point operations in an instruction set.

Another important instruction set measurement is the range of values for immediates. Like displacement values, the size of immediate values affects instruction length. As Figure A.10 shows, small immediate values are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations.

Summary: Memory Addressing

First, because of their popularity, we would expect a new architecture to support at least the following addressing modes: displacement, immediate, and register indirect. Figure A.7 shows that they represent 75%–99% of the addressing modes used

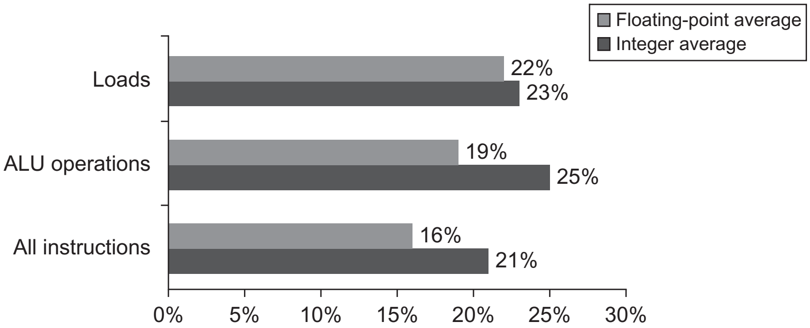


Figure A.9 About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about one-sixth of the instructions. For loads, the load immediate instruction loads 16 bits into either half of a 32-bit register. Load immediates are not loads in a strict sense because they do not access memory. Occasionally a pair of load immediates is used to load a 32-bit constant, but this is rare. (For ALU operations, shifts by a constant amount are included as operations with immediate operands.) The programs and computer used to collect these statistics are the same as in Figure A.8.

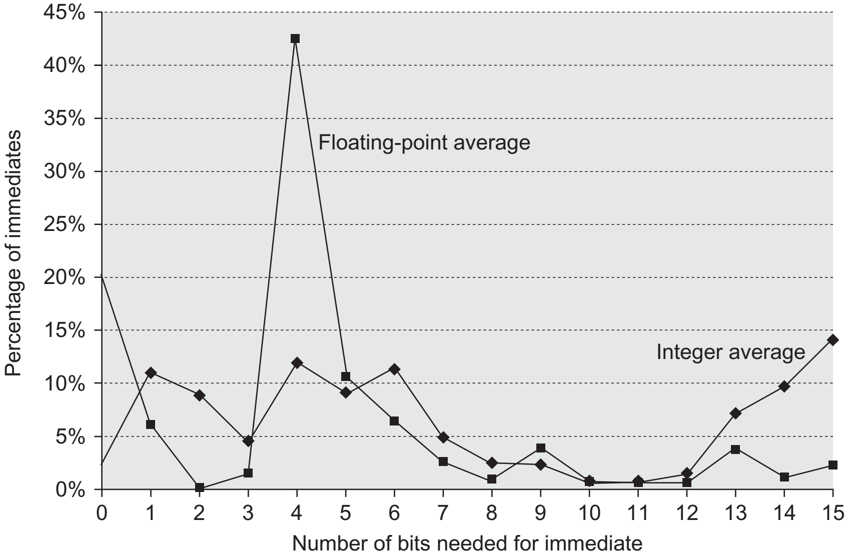


Figure A.10 The distribution of immediate values. The *x*-axis shows the number of bits needed to represent the magnitude of an immediate value—0 means the immediate field value was 0. The majority of the immediate values are positive. About 20% were negative for CINT2000, and about 30% were negative for CFP2000. These measurements were taken on an Alpha, where the maximum immediate is 16 bits, for the same programs as in Figure A.8. A similar measurement on the VAX, which supported 32-bit immediates, showed that about 20%–25% of immediates were longer than 16 bits. Thus, 16 bits would capture about 80% and 8 bits about 50%.

in our measurements. Second, we would expect the size of the address for displacement mode to be at least 12–16 bits, because the caption in Figure A.8 suggests these sizes would capture 75%–99% of the displacements. Third, we would expect the size of the immediate field to be at least 8–16 bits. This claim is not substantiated by the caption of the figure to which it refers.

Having covered instruction set classes and decided on register-register architectures, plus the previous recommendations on data addressing modes, we next cover the sizes and meanings of data.

A.4 Type and Size of Operands

How is the type of an operand designated? Usually, encoding in the opcode designates the type of an operand—this is the method used most often. Alternatively, the data can be annotated with tags that are interpreted by the hardware. These tags specify the type of the operand, and the operation is chosen accordingly. Computers with tagged data, however, can only be found in computer museums.

Let's start with desktop and server architectures. Usually the type of an operand—integer, single-precision floating point, character, and so on—effectively gives its size. Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words). Integers are almost universally represented as two's complement binary numbers. Characters are usually in ASCII, but the 16-bit Unicode (used in Java) is gaining popularity with the internationalization of computers. Until the early 1980s, most computer manufacturers chose their own floating-point representation. Almost all computers since that time follow the same standard for floating point, the IEEE standard 754, although this level of accuracy has recently been abandoned in application-specific processors. The IEEE floating-point standard is discussed in detail in Appendix J.

Some architectures provide operations on character strings, although such operations are usually quite limited and treat each byte in the string as a single character. Typical operations supported on character strings are comparisons and moves.

For business applications, some architectures support a decimal format, usually called *packed decimal* or *binary-coded decimal*—4 bits are used to encode the values 0–9, and 2 decimal digits are packed into each byte. Numeric character strings are sometimes called *unpacked decimal*, and operations—called *packing* and *unpacking*—are usually provided for converting back and forth between them.

One reason to use decimal operands is to get results that exactly match decimal numbers, as some decimal fractions do not have an exact representation in binary. For example, 0.10_{10} is a simple fraction in decimal, but in binary it requires an infinite set of repeating digits: $0.000110011\overline{0011}_{\ldots 2}$. Thus, calculations that are exact in decimal can be close but inexact in binary, which can be a problem for financial transactions. (See Appendix J to learn more about precise arithmetic.)

The SPEC benchmarks use byte or character, half-word (short integer), word (integer and single precision floating point), double-word (long integer), and floating-point data types. Figure A.11 shows the dynamic distribution of the sizes of objects referenced from memory for these programs. The frequency of access to

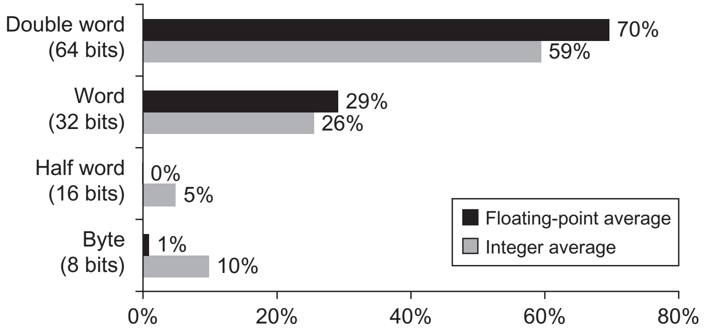


Figure A.11 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, because the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.

different data types helps in deciding what types are most important to support efficiently. Should the computer have a 64-bit access path, or would taking two cycles to access a double word be satisfactory? As we saw earlier, byte accesses require an alignment network: how important is it to support bytes as primitives? Figure A.11 uses memory references to examine the types of data being accessed.

In some architectures, objects in registers may be accessed as bytes or half words. However, such access is very infrequent—on the VAX, it accounts for no more than 12% of register references, or roughly 6% of all operand accesses in these programs.

A.5 Operations in the Instruction Set

The operators supported by most instruction set architectures can be categorized as in Figure A.12. One rule of thumb across all architectures is that the most widely executed instructions are the simple operations of an instruction set. For example, Figure A.13 shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on the popular Intel 80x86. Hence, the implementor of these instructions should be sure to make these fast, as they are the common case.

Operator type	**Examples**
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

Figure A.12 Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any computer that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel—for example, performing eight 8-bit additions on two 64-bit operands.

Rank	80x86 instruction	Integer average % total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		**96%**

Figure A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

As mentioned before, the instructions in Figure A.13 are found in every computer for every application—desktop, server, embedded—with the variations of operations in Figure A.12 largely depending on which data types the instruction set includes.

A.6 Instructions for Control Flow

Because the measurements of branch and jump behavior are fairly independent of other measurements and applications, we now examine the use of control flow instructions, which have little in common with the operations of the previous sections.

There is no consistent terminology for instructions that change the flow of control. In the 1950s they were typically called *transfers*. Beginning in 1960 the name *branch* began to be used. Later, computers introduced additional names. Throughout this book we will use *jump* when the change in control is unconditional and *branch* when the change is conditional.

We can distinguish four different types of control flow change:

- Conditional branches
- Jumps
- Procedure calls
- Procedure returns

We want to know the relative frequency of these events, as each event is different, may use different instructions, and may have different behavior. Figure A.14 shows the frequencies of these control flow instructions for a load-store computer running our benchmarks.

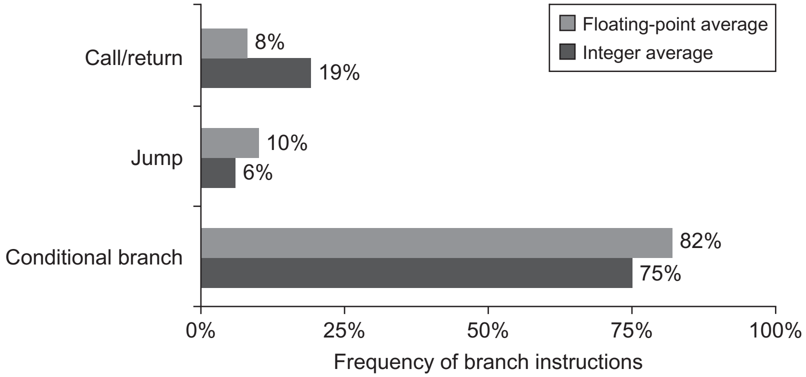


Figure A.14 Breakdown of control flow instructions into three classes: calls or returns, jumps, and conditional branches. Conditional branches clearly dominate. Each type is counted in one of three bars. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Addressing Modes for Control Flow Instructions

The destination address of a control flow instruction must always be specified. This destination is specified explicitly in the instruction in the vast majority of cases—procedure return being the major exception, because for return the target is not known at compile time. The most common way to specify the destination is to supply a displacement that is added to the *program counter* (PC). Control flow instructions of this sort are called *PC-relative*. PC-relative branches or jumps are advantageous because the target is often near the current instruction, and specifying the position relative to the current PC requires fewer bits. Using PC-relative addressing also permits the code to run independently of where it is loaded. This property, called *position independence*, can eliminate some work when the program is linked and is also useful in programs linked dynamically during execution.

To implement returns and indirect jumps when the target is not known at compile time, a method other than PC-relative addressing is required. Here, there must be a way to specify the target dynamically, so that it can change at runtime. This dynamic address may be as simple as naming a register that contains the target address; alternatively, the jump may permit any addressing mode to be used to supply the target address.

These register indirect jumps are also useful for four other important features:

- *Case* or *switch* statements, found in most programming languages (which select among one of several alternatives).
- *Virtual functions* or *methods* in object-oriented languages like C++ or Java (which allow different routines to be called depending on the type of the argument).

- *High-order functions* or *function pointers* in languages like C or C++ (which allow functions to be passed as arguments, giving some of the flavor of object-oriented programming).
- *Dynamically shared libraries* (which allow a library to be loaded and linked at runtime only when it is actually invoked by the program rather than loaded and linked statically before the program is run).

In all four cases the target address is not known at compile time, and hence is usually loaded from memory into a register before the register indirect jump.

As branches generally use PC-relative addressing to specify their targets, an important question concerns how far branch targets are from branches. Knowing the distribution of these displacements will help in choosing what branch offsets to support, and thus will affect the instruction length and encoding. Figure A.15 shows the distribution of displacements for PC-relative branches in instructions. About 75% of the branches are in the forward direction.

Conditional Branch Options

Because most changes in control flow are branches, deciding how to specify the branch condition is important. Figure A.16 shows the three primary techniques in use today and their advantages and disadvantages.

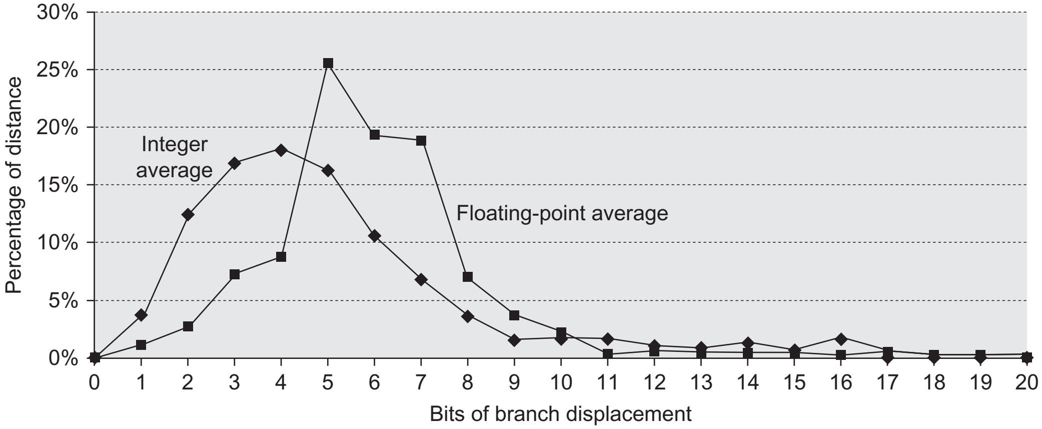


Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4–8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Figure A.16 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a separate operation and register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable because the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

One of the most noticeable properties of branches is that a large number of the comparisons are simple tests, and a large number are comparisons with zero. Thus, some architectures choose to treat these comparisons as special cases, especially if a *compare and branch* instruction is being used. Figure A.17 shows the frequency of different comparisons used for conditional branching.

Procedure Invocation Options

Procedure calls and returns include control transfer and possibly some state saving; at a minimum the return address must be saved somewhere, sometimes in a special link register or just a GPR. Some older architectures provide a mechanism to save many registers, while newer architectures require the compiler to generate stores and loads for each register saved and restored.

There are two basic conventions in use to save registers: either at the call site or inside the procedure being called. *Caller saving* means that the calling procedure must save the registers that it wants preserved for access after the call, and thus the called procedure need not worry about registers. *Callee saving* is the opposite: the called procedure must save the registers it wants to use, leaving the caller unrestrained. There are times when caller save must be used because of access patterns to globally visible variables in two different procedures. For example, suppose we have a procedure P1 that calls procedure P2, and both procedures manipulate the

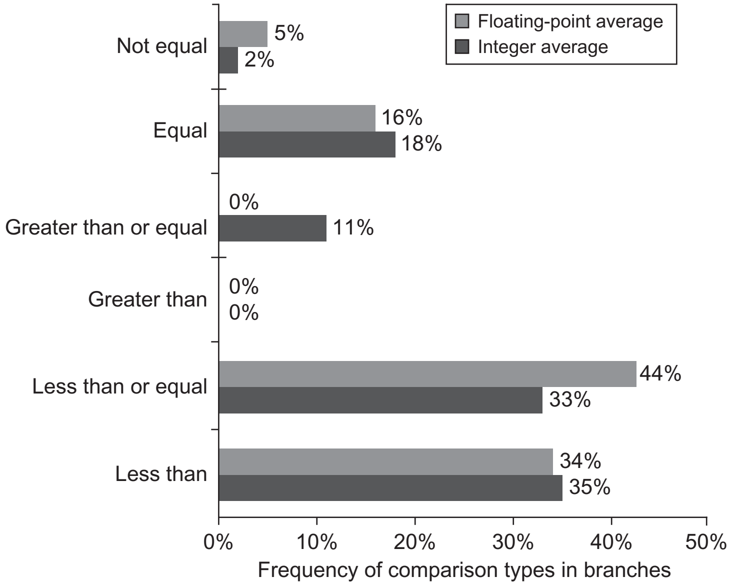


Figure A.17 Frequency of different types of compares in conditional branches. Less than (or equal) branches dominate this combination of compiler and architecture. These measurements include both the integer and floating-point compares in branches. The programs and computer used to collect these statistics are the same as those in Figure A.8.

global variable *x*. If P1 had allocated *x* to a register, it must be sure to save *x* to a location known by P2 before the call to P2. A compiler's ability to discover when a called procedure may access register-allocated quantities is complicated by the possibility of separate compilation. Suppose P2 may not touch *x* but can call another procedure, P3, that may access *x*, yet P2 and P3 are compiled separately. Because of these complications, most compilers will conservatively caller save *any* variable that may be accessed during a call.

In the cases where either convention could be used, some programs will be more optimal with callee save and some will be more optimal with caller save. As a result, most real systems today use a combination of the two mechanisms. This convention is specified in an application binary interface (ABI) that sets down the basic rules as to which registers should be caller saved and which should be callee saved. Later in this appendix we will examine the mismatch between sophisticated instructions for automatically saving registers and the needs of the compiler.

Summary: Instructions for Control Flow

Control flow instructions are some of the most frequently executed instructions. Although there are many options for conditional branches, we would expect branch addressing in a new architecture to be able to jump to hundreds of instructions either above or below the branch. This requirement suggests a PC-relative branch displacement of at least 8 bits. We would also expect to see register indirect and PC-relative addressing for jump instructions to support returns as well as many other features of current systems.

We have now completed our instruction architecture tour at the level seen by an assembly language programmer or compiler writer. We are leaning toward a load-store architecture with displacement, immediate, and register indirect addressing modes. These data are 8-, 16-, 32-, and 64-bit integers and 32- and 64-bit floating-point data. The instructions include simple operations, PC-relative conditional branches, jump and link instructions for procedure call, and register indirect jumps for procedure return (plus a few other uses).

Now we need to select how to represent this architecture in a form that makes it easy for the hardware to execute.

A.7 Encoding an Instruction Set

Clearly, the choices mentioned herein will affect how the instructions are encoded into a binary representation for execution by the processor. This representation affects not only the size of the compiled program but also the implementation of the processor, which must decode this representation to quickly find the operation and its operands. The operation is typically specified in one field, called the *opcode*. As we shall see, the important decision is how to encode the addressing modes with the operations.

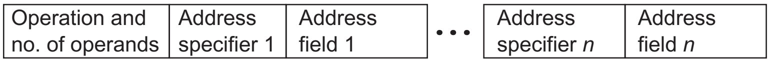
This decision depends on the range of addressing modes and the degree of independence between opcodes and modes. Some older computers have one to five operands with 10 addressing modes for each operand (see Figure A.6). For such a large number of combinations, typically a separate *address specifier* is needed for each operand: the address specifier tells what addressing mode is used to access the operand. At the other extreme are load-store computers with only one memory operand and only one or two addressing modes; obviously, in this case, the addressing mode can be encoded as part of the opcode.

When encoding the instructions, the number of registers and the number of addressing modes both have a significant impact on the size of instructions, as the register field and addressing mode field may appear many times in a single instruction. In fact, for most instructions many more bits are consumed in encoding addressing modes and register fields than in specifying the opcode. The architect must balance several competing forces when encoding the instruction set:

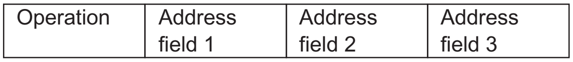
1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.

3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation. (The value of easily decoded instructions is discussed in Appendix C and Chapter 3.) As a minimum, the architect wants instructions to be in multiples of bytes, rather than an arbitrary bit length. Many desktop and server architects have chosen to use a fixed-length instruction to gain implementation benefits while sacrificing average code size.

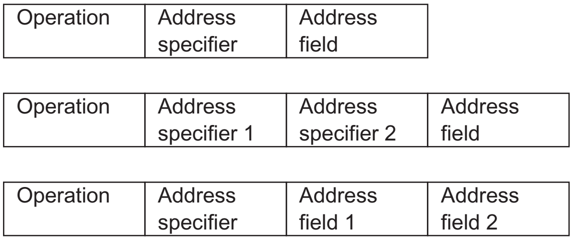
Figure A.18 shows three popular choices for encoding the instruction set. The first we call *variable*, because it allows virtually all addressing modes to be with all operations. This style is best when there are many addressing modes and operations. The second choice we call *fixed*, because it combines the operation and the addressing mode into the opcode. Often fixed encoding will have only a single


Operation and no. of operands
Address specifier 1
Address field 1
• • •
Address specifier *n*
Address field *n*

(A) Variable (e.g., Intel 80x86, VAX)


Operation
Address field 1
Address field 2
Address field 3

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)


Operation
Address specifier
Address field
Operation
Address specifier 1
Address specifier 2
Address field
Operation
Address specifier
Address field 1
Address field 2

(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, because unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode. It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address.

size for all instructions; it works best when there are few addressing modes and operations. The trade-off between variable encoding and fixed encoding is size of programs versus ease of decoding in the processor. Variable tries to use as few bits as possible to represent the program, but individual instructions can vary widely in both size and the amount of work to be performed.

Let's look at an 80x86 instruction to see an example of the variable encoding:

```
add EAX,1000(EBX)
```

The name `add` means a 32-bit integer add instruction with two operands, and this opcode takes 1 byte. An 80x86 address specifier is 1 or 2 bytes, specifying the source/destination register (`EAX`) and the addressing mode (displacement in this case) and base register (`EBX`) for the second operand. This combination takes 1 byte to specify the operands. When in 32-bit mode (see Appendix K), the size of the address field is either 1 byte or 4 bytes. Because1000 is bigger than 2^8, the total length of the instruction is

$$1 + 1 + 4 = 6 \text{ bytes}$$

The length of 80x86 instructions varies between 1 and 17 bytes. 80x86 programs are generally smaller than the RISC architectures, which use fixed formats (see Appendix K).

Given these two poles of instruction set design of variable and fixed, the third alternative immediately springs to mind: reduce the variability in size and work of the variable architecture but provide multiple instruction lengths to reduce code size. This *hybrid* approach is the third encoding alternative, and we'll see examples shortly.

Reduced Code Size in RISCs

As RISC computers started being used in embedded applications, the 32-bit fixed format became a liability because cost, and hence smaller code, are important. In response, several manufacturers offered a new hybrid version of their RISC instruction sets, with both 16-bit and 32-bit instructions. The narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and the two-address format rather than the classic three-address format of RISC computers. RISC-V offers such an extension, called RV32IC, the C standing for compressed. Common instruction occurrences, such as intermediates with small values and common ALU operations with the source and destination register being identical, are encoded in 16-bit formats. Appendix K gives two other examples, the ARM Thumb and microMIPS, which both claim a code size reduction of up to 40%.

In contrast to these instruction set extensions, IBM simply compresses its standard instruction set and then adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss. Thus, the instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk. The advantage of a compressed format, such as RV32IC, microMIPS and Thumb2 is that instruction caches act as if they are about 25% larger, while IBM's CodePack means that compilers need not be changed to handle different instruction sets and instruction decoding can remain simple.

CodePack starts with run-length encoding compression on any PowerPC program and then loads the resulting compression tables in a 2 KB table on chip. Hence, every program has its own unique encoding. To handle branches, which are no longer to an aligned word boundary, the PowerPC creates a hash table in memory that maps between compressed and uncompressed addresses. Like a TLB (see Chapter 2), it caches the most recently used address maps to reduce the number of memory accesses. IBM claims an overall performance cost of 10%, resulting in a code size reduction of 35%–40%.

Summary: Encoding an Instruction Set

Decisions made in the components of instruction set design discussed in previous sections determine whether the architect has the choice between variable and fixed instruction encodings. Given the choice, the architect more interested in code size than performance will pick variable encoding, and the one more interested in performance than code size will pick fixed encoding. RISC-V, MIPS, and ARM all have an instruction set extension that uses 16-bit instruction, as well as 32-bit; applications with serious code size constraints can opt to use the 16-bit variant to decrease code size. Appendix E gives 13 examples of the results of architects' choices. In Appendix C and Chapter 3, the impact of variability on performance of the processor will be discussed further.

We have almost finished laying the groundwork for the RISC-V instruction set architecture that will be introduced in Section A.9. Before we do that, however, it will be helpful to take a brief look at compiler technology and its effect on program properties.

A.8 Cross-Cutting Issues: The Role of Compilers

Today almost all programming is done in high-level languages for desktop and server applications. This development means that because most instructions executed are the output of a compiler, an instruction set architecture is essentially a compiler target. In earlier times for these applications, architectural decisions were often made to ease assembly language programming or for a specific kernel. Because the compiler will significantly affect the performance of a computer, understanding compiler technology today is critical to designing and efficiently implementing an instruction set.

Once it was popular to try to isolate the compiler technology and its effect on hardware performance from the architecture and its performance, just as it was popular to try to separate architecture from its implementation. This separation is essentially impossible with today's desktop compilers and computers. Architectural choices affect the quality of the code that can be generated for a computer and the complexity of building a good compiler for it, for better or for worse.

In this section, we discuss the critical goals in the instruction set primarily from the compiler viewpoint. It starts with a review of the anatomy of current compilers. Next we discuss how compiler technology affects the decisions of the architect, and how the architect can make it hard or easy for the compiler to produce good code. We conclude with a review of compilers and multimedia operations, which unfortunately is a bad example of cooperation between compiler writers and architects.

The Structure of Recent Compilers

To begin, let's look at what optimizing compilers are like today. Figure A.19 shows the structure of recent compilers.

A compiler writer's first goal is correctness—all valid programs must be compiled correctly. The second goal is usually speed of the compiled code. Typically, a whole set of other goals follows these two, including fast compilation, debugging support, and interoperability among languages. Normally, the passes

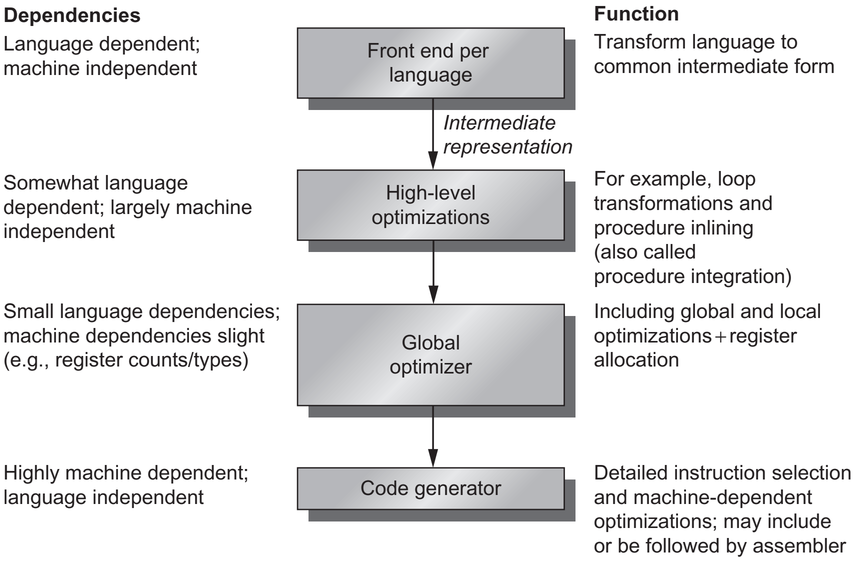

Dependencies
Language dependent; machine independent
Somewhat language dependent; largely machine independent
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Highly machine dependent; language independent
Front end per language
Intermediate representation
High-level optimizations
Global optimizer
Code generator
Function
Transform language to common intermediate form
For example, loop transformations and procedure inlining (also called procedure integration)
Including global and local optimizations + register allocation
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

in the compiler transform higher-level, more abstract representations into progressively lower-level representations. Eventually it reaches the instruction set. This structure helps manage the complexity of the transformations and makes writing a bug-free compiler easier.

The complexity of writing a correct compiler is a major limitation on the amount of optimization that can be done. Although the multiple-pass structure helps reduce compiler complexity, it also means that the compiler must order and perform some transformations before others. In the diagram of the optimizing compiler in Figure A.19, we can see that certain high-level optimizations are performed long before it is known what the resulting code will look like. Once such a transformation is made, the compiler can't afford to go back and revisit all steps, possibly undoing transformations. Such iteration would be prohibitive, both in compilation time and in complexity. Thus, compilers make assumptions about the ability of later steps to deal with certain problems. For example, compilers usually have to choose which procedure calls to expand inline before they know the exact size of the procedure being called. Compiler writers call this problem the *phase-ordering problem*.

How does this ordering of transformations interact with the instruction set architecture? A good example occurs with the optimization called *global common subexpression elimination*. This optimization finds two instances of an expression that compute the same value and saves the value of the first computation in a temporary. It then uses the temporary value, eliminating the second computation of the common expression.

For this optimization to be significant, the temporary must be allocated to a register. Otherwise, the cost of storing the temporary in memory and later reloading it may negate the savings gained by not recomputing the expression. There are, in fact, cases where this optimization actually slows down code when the temporary is not register allocated. Phase ordering complicates this problem because register allocation is typically done near the end of the global optimization pass, just before code generation. Thus, an optimizer that performs this optimization must *assume* that the register allocator will allocate the temporary to a register.

Optimizations performed by modern compilers can be classified by the style of the transformation, as follows:

- *High-level optimizations* are often done on the source with output fed to later optimization passes.
- *Local optimizations* optimize code only within a straight-line code fragment (called a *basic block* by compiler people).
- *Global optimizations* extend the local optimizations across branches and introduce a set of transformations aimed at optimizing loops.
- *Register allocation* associates registers with operands.
- *Processor-dependent optimizations* attempt to take advantage of specific architectural knowledge.

Register Allocation

Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important—if not the most important—of the optimizations. Register allocation algorithms today are based on a technique called *graph coloring*. The basic idea behind graph coloring is to construct a graph representing the possible candidates for allocation to a register and then to use the graph to allocate registers. Roughly speaking, the problem is how to use a limited set of colors so that no two adjacent nodes in a dependency graph have the same color. The emphasis in the approach is to achieve 100% register allocation of active variables. The problem of coloring a graph in general can take exponential time as a function of the size of the graph (NP-complete). There are heuristic algorithms, however, that work well in practice, yielding close allocations that run in near-linear time.

Graph coloring works best when there are at least 16 (and preferably more) general-purpose registers available for global allocation for integer variables and additional registers for floating point. Unfortunately, graph coloring does not work very well when the number of registers is small because the heuristic algorithms for coloring the graph are likely to fail.

Impact of Optimizations on Performance

It is sometimes difficult to separate some of the simpler optimizations—local and processor-dependent optimizations—from transformations done in the code generator. Examples of typical optimizations are given in Figure A.20. The last column of Figure A.20 indicates the frequency with which the listed optimizing transforms were applied to the source program.

Figure A.21 shows the effect of various optimizations on instructions executed for two programs. In this case, optimized programs executed roughly 25%–90% fewer instructions than unoptimized programs. The figure illustrates the importance of looking at optimized code before suggesting new instruction set features, because a compiler might completely remove the instructions the architect was trying to improve.

The Impact of Compiler Technology on the Architect's Decisions

The interaction of compilers and high-level languages significantly affects how programs use an instruction set architecture. There are two important questions: how are variables allocated and addressed? How many registers are needed to allocate variables appropriately? To address these questions, we must look at the three separate areas in which current high-level languages allocate their data:

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., $A = X$) with *X*	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

Figure A.20 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. The third column lists the static frequency with which some of the common optimizations are applied in a set of 12 small Fortran and Pascal programs. There are nine local and global optimizations done by the compiler included in the measurement. Six of these optimizations are covered in the figure, and the remaining three account for 18% of the total static occurrences. The abbreviation *N.M.* means that the number of occurrences of that optimization was not measured. Processor-dependent optimizations are usually done in a code generator, and none of those was measured in this experiment. The percentage is the portion of the static optimizations that are of the specified type. Data from Chow, F.C., 1983. A Portable Machine-Independent Global Optimizer—Design and Measurements (Ph.D. thesis). Stanford University, Palo Alto, CA (collected using the Stanford UCODE compiler).

- The *stack* is used to allocate local variables. The stack is grown or shrunk on procedure call or return, respectively. Objects on the stack are addressed relative to the stack pointer and are primarily scalars (single variables) rather than arrays. The stack is used for activation records, *not* as a stack for evaluating expressions. Hence, values are almost never pushed or popped on the stack.

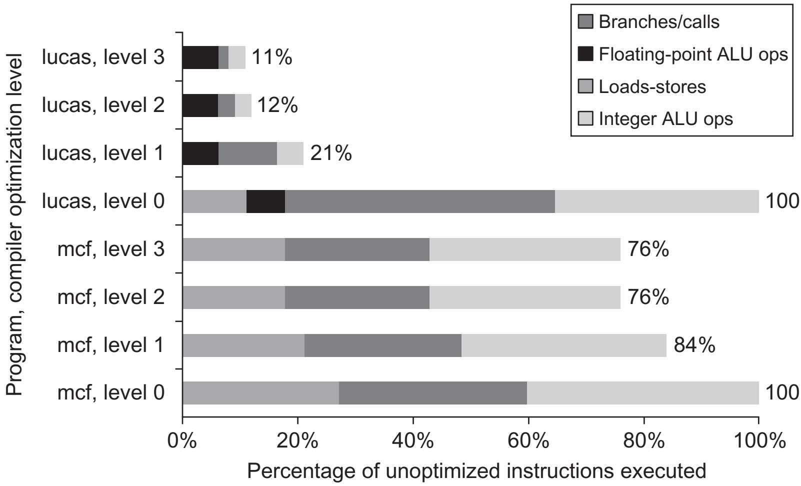


Figure A.21 Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration. These experiments were performed on Alpha compilers.

- The *global data area* is used to allocate statically declared objects, such as global variables and constants. A large percentage of these objects are arrays or other aggregate data structures.
- The *heap* is used to allocate dynamic objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers and are typically not scalars.

Register allocation is much more effective for stack-allocated objects than for global variables, and register allocation is essentially impossible for heap-allocated objects because they are accessed with pointers. Global variables and some stack variables are impossible to allocate because they are *aliased*—there are multiple ways to refer to the address of a variable, making it illegal to put it into a register. (Most heap variables are effectively aliased for today's compiler technology.)

For example, consider the following code sequence, where `&` returns the address of a variable and `*` dereferences a pointer:

```
p =&a   - gets address of a in p
a =...   - assigns to a directly
*p =...   - uses p to assign to a
...a...   - accesses a
```

The variable `a` could not be register allocated across the assignment to `*p` without generating incorrect code. Aliasing causes a substantial problem because it is often difficult or impossible to decide what objects a pointer may refer to. A compiler must be conservative; some compilers will not allocate *any* local variables of a procedure in a register when there is a pointer that may refer to *one* of the local variables.

How the Architect Can Help the Compiler Writer

Today, the complexity of a compiler does not come from translating simple statements like A = B + C. Most programs are *locally simple*, and simple translations work fine. Rather, complexity arises because programs are large and globally complex in their interactions, and because the structure of compilers means decisions are made one step at a time about which code sequence is best.

Compiler writers often are working under their own corollary of a basic principle in architecture: *make the frequent cases fast and the rare case correct*. That is, if we know which cases are frequent and which are rare, and if generating code for both is straightforward, then the quality of the code for the rare case may not be very important—but it must be correct!

Some instruction set properties help the compiler writer. These properties should not be thought of as hard-and-fast rules, but rather as guidelines that will make it easier to write a compiler that will generate efficient and correct code.

- *Provide regularity*—Whenever it makes sense, the three primary components of an instruction set—the operations, the data types, and the addressing modes—should be *orthogonal*. Two aspects of an architecture are said to be orthogonal if they are independent. For example, the operations and addressing modes are orthogonal if, for every operation to which one addressing mode can be applied, all addressing modes are applicable. This regularity helps simplify code generation and is particularly important when the decision about what code to generate is split into two passes in the compiler. A good counterexample of this property is restricting what registers can be used for a certain class of instructions. Compilers for special-purpose register architectures typically get stuck in this dilemma. This restriction can result in the compiler finding itself with lots of available registers, but none of the right kind!
- *Provide primitives, not solutions*—Special features that "match" a language construct or a kernel function are often unusable. Attempts to support high-level languages may work only with one language or do more or less than is required for a correct and efficient implementation of the language. An example of how such attempts have failed is given in Section A.10.
- *Simplify trade-offs among alternatives*—One of the toughest jobs a compiler writer has is figuring out what instruction sequence will be best for every segment of code that arises. In earlier days, instruction counts or total code size might have been good metrics, but—as we saw in Chapter 1—this is no longer

true. With caches and pipelining, the trade-offs have become very complex. Anything the designer can do to help the compiler writer understand the costs of alternative code sequences would help improve the code. One of the most difficult instances of complex trade-offs occurs in a register-memory architecture in deciding how many times a variable should be referenced before it is cheaper to load it into a register. This threshold is hard to compute and, in fact, may vary among models of the same architecture.

- *Provide instructions that bind the quantities known at compile time as constants*—A compiler writer hates the thought of the processor interpreting at runtime a value that was known at compile time. Good counterexamples of this principle include instructions that interpret values that were fixed at compile time. For instance, the VAX procedure call instruction (`calls`) dynamically interprets a mask saying what registers to save on a call, but the mask is fixed at compile time (see Section A.10).

Compiler Support (or Lack Thereof) for Multimedia Instructions

Alas, the designers of the SIMD instructions (see Section 4.3 in Chapter 4) basically ignored the previous subsection. These instructions tend to be solutions, not primitives; they are short of registers; and the data types do not match existing programming languages. Architects hoped to find an inexpensive solution that would help some users, but often only a few low-level graphics library routines use them.

The SIMD instructions are really an abbreviated version of an elegant architecture style that has its own compiler technology. As explained in Section 4.2, *vector architectures* operate on vectors of data. Invented originally for scientific codes, multimedia kernels are often vectorizable as well, albeit often with shorter vectors. As Section 4.3 suggests, we can think of Intel's MMX and SSE or PowerPC's AltiVec, or the RISC-V P extension, as simply short vector computers: MMX with vectors of eight 8-bit elements, four 16-bit elements, or two 32-bit elements, and AltiVec with vectors twice that length. They are implemented as simply adjacent, narrow elements in wide registers.

These microprocessor architectures build the vector register size into the architecture: the sum of the sizes of the elements is limited to 64 bits for MMX and 128 bits for AltiVec. When Intel decided to expand to 128-bit vectors, it added a whole new set of instructions, called streaming SIMD extension (SSE).

A major advantage of vector computers is hiding latency of memory access by loading many elements at once and then overlapping execution with data transfer. The goal of vector addressing modes is to collect data scattered about memory, place them in a compact form so that they can be operated on efficiently, and then place the results back where they belong.

Vector computers include *strided addressing* and *gather/scatter addressing* (see Section 4.2) to increase the number of programs that can be vectorized. Strided addressing skips a fixed number of words between each access, so sequential addressing is often called *unit stride addressing*. Gather and scatter find their addresses in another vector register: think of it as register indirect addressing for vector computers. From a vector perspective, in contrast, these short-vector SIMD computers support only unit strided accesses: memory accesses load or store all elements at once from a single wide memory location. Because the data for multimedia applications are often streams that start and end in memory, strided and gather/scatter addressing modes are essential to successful vectorization (see Section 4.7).

Example As an example, compare a vector computer with MMX for color representation conversion of pixels from RGB (red, green, blue) to YUV (luminosity chrominance), with each pixel represented by 3 bytes. The conversion is just three lines of C code placed in a loop:

```
Y =(9798*R +19235*G +3736*B) / 32768;
U =(-4784*R 9437*G +4221*B) / 32768 +128;
V =(20218*R 16941*G 3277*B) / 32768 +128;
```

A 64-bit-wide vector computer can calculate 8 pixels simultaneously. One vector computer for media with strided addresses takes

- 3 vector loads (to get RGB)
- 3 vector multiplies (to convert R)
- 6 vector multiply adds (to convert G and B)
- 3 vector shifts (to divide by 32,768)
- 2 vector adds (to add 128)
- 3 vector stores (to store YUV)

The total is 20 instructions to perform the 20 operations in the previous C code to convert 8 pixels (Kozyrakis, 2000). (Because a vector might have 32 64-bit elements, this code actually converts up to 32 × 8 or 256 pixels.)

In contrast, Intel's website shows that a library routine to perform the same calculation on 8 pixels takes 116 MMX instructions plus 6 80x86 instructions (Intel, 2001). This six-fold increase in instructions is due to the large number of instructions to load and unpack RGB pixels and to pack and store YUV pixels, because there are no strided memory accesses.

Having short, architecture-limited vectors with few registers and simple memory addressing modes makes it more difficult to use vectorizing compiler technology. Hence, these SIMD instructions are more likely to be found in hand-coded libraries than in compiled code.

Summary: The Role of Compilers

This section leads to several recommendations. First, we expect a new instruction set architecture to have at least 16 general-purpose registers—not counting separate registers for floating-point numbers—to simplify allocation of registers using graph coloring. The advice on orthogonality suggests that all supported addressing modes apply to all instructions that transfer data. Finally, the last three pieces of advice—provide primitives instead of solutions, simplify trade-offs between alternatives, don't bind constants at runtime—all suggest that it is better to err on the side of simplicity. In other words, understand that less is more in the design of an instruction set. Alas, SIMD extensions are more an example of good marketing than of outstanding achievement of hardware–software co-design.

A.9 Putting It All Together: The RISC-V Architecture

In this section we describe the load-store architecture called RISC-V. RISC-V is a freely licensed open standard, similar to many of the RISC architectures, and based on observations similar to those covered in the last sections. (In Section M.3 we discuss how and why these architectures became popular.) RISC-V builds on 30 years of experience with RISC architectures and "cleans up" most of the short-term inclusions and omissions, leading to an architecture that is easier and more efficient to implement. RISC-V provides a both a 32-bit and a 64-bit instruction set, as well as a variety of extensions for features like floating point; these extensions can be added to either the 32-bit or 64-bit base instruction set. We discuss a 64-bit version of RISC-V, RV64, which is a superset of the 32-bit version RV32.

Reviewing our expectations from each section, for desktop and server applications:

- Section A.2—Use general-purpose registers with a load-store architecture.
- Section A.3—Support these addressing modes: displacement (with an address offset size of 12–16 bits), immediate (size 8–16 bits), and register indirect.
- Section A.4—Support these data sizes and types: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
- Section A.5—Support these simple instructions, because they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and shift.
- Section A.6—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- Section A.7—Use fixed instruction encoding if interested in performance, and use variable instruction encoding if interested in code size. In some low-end, embedded applications, with small or only one-level caches, larger code size may have significant performance implications. ISAs that provide a compressed instruction set extension provide a way of addressing this difference.
- Section A.8—Provide at least 16, and preferably 32, general-purpose registers, be sure all addressing modes apply to all data transfer instructions, and aim for

a minimalist instruction set. This section didn't cover floating-point programs, but they often use separate floating-point registers. The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.

We introduce RISC-V by showing how it follows these recommendations. Like its RISC predecessors, RISC-V emphasizes

- A simple load-store instruction set.
- Design for pipelining efficiency (discussed in Appendix C), including a fixed instruction set encoding.
- Efficiency as a compiler target.

RISC-V provides a good architectural model for study, not only because of the popularity of this type of processor, but also because it is an easy architecture to understand. We will use this architecture again in Appendix C and in Chapter 3, and it forms the basis for a number of exercises and programming projects.

RISC-V Instruction Set Organization

The RISC-V instruction set is organized as three base instruction sets that support 32-bit or 64-bit integers, and a variety of optional extensions to one of the base instruction sets. This allows RISC-V to be implemented for a wide range of potential applications from a small embedded processor with a minimal budget for logic and memory that likely costs $1 or less, to high-end processor configurations with full support for floating point, vectors, and multiprocessor configurations. Figure A.22 summarizes the three base instruction sets and the instruction set extensions with their basic functionality. For purposes of this text, we use RV64IMAFD (also known as RV64G, for short) in examples. RV32G is the 32-bit subset of the 64-bit architecture RV64G.

Registers for RISC-V

RV64G has 32 64-bit general-purpose registers (GPRs), named x0, x1, … , x31. GPRs are also sometimes known as *integer registers*. Additionally, with the F and D extensions for floating point that are part of RV64G, come a set of 32 floating-point registers (FPRs), named f0, f1, … , f31, which can hold 32 single-precision (32-bit) values or 32 double-precision (64-bit) values. (When holding one single-precision number, the other half of the FPR is unused.) Both single- and double-precision floating-point operations (32-bit and 64-bit) are provided.

The value of x0 is always 0. We shall see later how we can use this register to synthesize a variety of useful operations from a simple instruction set.

A few special registers can be transferred to and from the general-purpose registers. An example is the floating-point status register, used to hold information

Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

Figure A.22 RISC-V has three base instructions sets (and a reserved spot for a future fourth); all the extensions extend one of the base instruction sets. An instruction set is thus named by the base name followed by the extensions. For example, RISC-V64IMAFD refers to the base 64-bit instruction set with extensions M, A, F, and D. For consistency of naming and software, this combination is given the abbreviated name: RV64G, and we use RV64G through most of this text.

about the results of floating-point operations. There are also instructions for moving between an FPR and a GPR.

Data Types for RISC-V

The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit doublewords for integer data and 32-bit single precision and 64-bit double precision for floating point. Half words were added because they are found in languages like C and are popular in some programs, such as the operating systems, concerned about size of data structures. They will also become more popular if Unicode becomes widely used.

The RV64G operations work on 64-bit integers and 32- or 64-bit floating point. Bytes, half words, and words are loaded into the general-purpose registers with either zeros or the sign bit replicated to fill the 64 bits of the GPRs. Once loaded, they are operated on with the 64-bit integer operations.

Addressing Modes for RISC-V Data Transfers

The only data addressing modes are immediate and displacement, both with 12-bit fields. Register indirect is accomplished simply by placing 0 in the 12-bit displacement field, and limited absolute addressing with a 12-bit field is accomplished by using register 0 as the base register. Embracing zero gives us four effective modes, although only two are supported in the architecture.

RV64G memory is byte addressable with a 64-bit address and uses Little Endian byte numbering. As it is a load-store architecture, all references between memory and either GPRs or FPRs are through loads or stores. Supporting the data types mentioned herein, memory accesses involving GPRs can be to a byte, half word, word, or double word. The FPRs may be loaded and stored with single-precision or double-precision numbers. Memory accesses need not be aligned; however, it may be that unaligned accesses run extremely slow. In practice, programmers and compilers would be stupid to use unaligned accesses.

RISC-V Instruction Format

Because RISC-V has just two addressing modes, these can be encoded into the opcode. Following the advice on making the processor easy to pipeline and decode, all instructions are 32 bits with a 7-bit primary opcode. Figure A.23 shows the instruction layout of the four major instruction types. These formats are simple while providing 12-bit fields for displacement addressing, immediate constants, or PC-relative branch addresses.

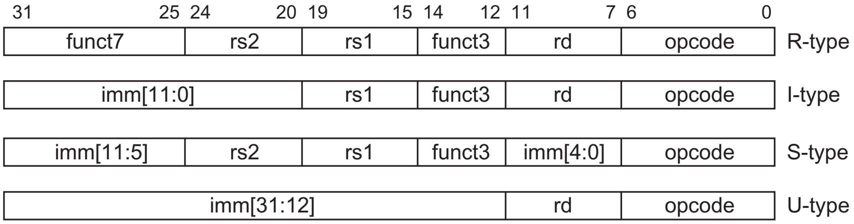


Figure A.23 The RISC-V instruction layout. There are two variations on these formata, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Figure A.24 The use of instruction fields for each instruction type. Primary use shows the major instructions that use the format. A blank indicates that the corresponding field is not present in this instruction type. The I-format is used for both loads and ALU immediates, with the 12-bit immediate holding either the value for an immediate or the displacement for a load. Similarly, the S-format encodes both store instructions (where the first source register is the base register and the second contains the register source for the value to store) and compare and branch instructions (where the register fields contain the sources to compare and the immediate field specifies the offset of the branch target). There are actually two other formats: SB and UJ that follow the same basic organization as S and J, but slightly modify the interpretation of the immediate fields.

The instruction formats and the use of the instruction fields is described in Figure A.24. The opcode specifies the general instruction type (ALU instruction, ALU immediate, load, store, branch, or jump), while the funct fields are used for specific operations. For example, an ALU instruction is encoded with a single opcode with the funct field dictating the exact operation: add, subtract, and, etc. Notice that several formats encode multiple types of instructions, including the use of the I-format for both ALU immediates and loads, and the use of the S-format for stores and conditional branches.

RISC-V Operations

RISC-V (or more properly RV64G) supports the list of simple operations recommended herein plus a few others. There are four broad classes of instructions: loads and stores, ALU operations, branches and jumps, and floating-point operations.

Any of the general-purpose or floating-point registers may be loaded or stored, except that loading x0 has no effect. Figure A.25 gives examples of the load and store instructions. Single-precision floating-point numbers occupy half a floating-point register. Conversions between single and double precision must be done explicitly. The floating-point format is IEEE 754 (see Appendix J). A list of all the RV64G instructions appears in Figure A.28 (page A.42).

Example instruction	Instruction name	Meaning
ld x1,80(x2)	Load doubleword	Regs[x1]←Mem[80+Regs[x2]]
lw x1,60(x2)	Load word	Regs[x1]$\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
lwu x1,60(x2)	Load word unsigned	Regs[x1]$\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
lb x1,40(x3)	Load byte	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
lbu x1,40(x3)	Load byte unsigned	Regs[x1]$\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
lh x1,40(x3)	Load half word	Regs[x1]$\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
flw f0,50(x3)	Load FP single	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
fld f0,50(x2)	Load FP double	Regs[f0]$\leftarrow_{64}$ Mem[50+Regs[x2]]
sd x2,400(x3)	Store double	Mem[400+Regs[x3]]$\leftarrow_{64}$ Regs[x2]
sw x3,500(x4)	Store word	Mem[500+Regs[x4]]$\leftarrow_{32}$ Regs[x3]$_{32..63}$
fsw f0,40(x3)	Store FP single	Mem[40+Regs[x3]]$\leftarrow_{32}$ Regs[f0]$_{0..31}$
fsd f0,40(x3)	Store FP double	Mem[40+Regs[x3]]$\leftarrow_{64}$ Regs[f0]
sh x3,502(x2)	Store half	Mem[502+Regs[x2]]$\leftarrow_{16}$ Regs[x3]$_{48..63}$
sb x2,41(x3)	Store byte	Mem[41+Regs[x3]]$\leftarrow_{8}$ Regs[x2]$_{56..63}$

Figure A.25 The load and store instructions in RISC-V. Loads shorter than 64 bits are available in both sign-extended and zero-extended forms. All memory references use a single addressing mode. Of course, both loads and stores are available for all the data types shown. Because RV64G supports double precision floating point, all single precision floating point loads must be aligned in the FP register, which are 64-bits wide.

To understand these figures we need to introduce a few additional extensions to our C description language used initially on page A-9:

- A subscript is appended to the symbol ← whenever the length of the datum being transferred might not be clear. Thus, $\leftarrow_n$ means transfer an n-bit quantity. We use x, $y \leftarrow z$ to indicate that z should be transferred to x and y.
- A subscript is used to indicate selection of a bit from a field. Bits are labeled from the most-significant bit starting at 0. The subscript may be a single digit (e.g., Regs[x4]$_0$ yields the sign bit of x4) or a subrange (e.g., Regs [x3]$_{56..63}$ yields the least-significant byte of x3).
- The variable Mem, used as an array that stands for main memory, is indexed by a byte address and may transfer any number of bytes.
- A superscript is used to replicate a field (e.g., 0^{48} yields a field of zeros of length 48 bits).
- The symbol ## is used to concatenate two fields and may appear on either side of a data transfer, and the symbols $\ll$ and $\gg$ shift the first operand left or right by the amount of the second operand.

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	Regs[x1]←Regs[x2]+Regs[x3]
`addi x1,x2,3`	Add immediate unsigned	Regs[x1]←Regs[x2]+3
`lui x1,42`	Load upper immediate	Regs[x1]←0^{32}##42##0^{12}
`sll x1,x2,5`	Shift left logical	Regs[x1]←Regs[x2]<<5
`slt x1,x2,x3`	Set less than	if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0

Figure A.26 The basic ALU instructions in RISC-V are available both with register-register operands and with one immediate operand. LUI uses the U-format that employs the rs1 field as part of the immediate, yielding a 20-bit immediate.

As an example, assuming that x8 and x10 are 32-bit registers:

$$\texttt{Regs[x10]}\leftarrow_{64}(\texttt{Mem[Regs[x8]]}_0)^{32}\#\#\ \texttt{Mem[Regs[R8]]}$$

means that the word at the memory location addressed by the contents of register x8 is sign-extended to form a 64-bit quantity that is stored into register x10.

All ALU instructions are register-register instructions. Figure A.26 gives some examples of the arithmetic/logical instructions. The operations include simple arithmetic and logical operations: add, subtract, AND, OR, XOR, and shifts. Immediate forms of all these instructions are provided using a 12-bit sign-extended immediate. The operation `LUI` (load upper immediate) loads bits 12–31 of a register, sign-extends the immediate field to the upper 32-bits, and sets the low-order 12-bits of the register to 0. `LUI` allows a 32-bit constant to be built in two instructions, or a data transfer using any constant 32-bit address in one extra instruction.

As mentioned herein, x0 is used to synthesize popular operations. Loading a constant is simply an add immediate where the source operand is x0, and a register-register move is simply an add (or an or) where one of the sources is x0. (We sometimes use the mnemonic li, standing for load immediate, to represent the former, and the mnemonic mv for the latter.)

RISC-V Control Flow Instructions

Control is handled through a set of jumps and a set of branches, and Figure A.27 gives some typical branch and jump instructions. The two jump instructions (jump and link and jump and link register) are unconditional transfers and always store the “link,” which is the address of the instruction sequentially following the jump instruction, in the register specified by the rd field. In the event that the link address is not needed, the rd field can simply be set to x0, which results in a typical unconditional jump. The two jump instructions are differentiated by whether the address is computed by adding an immediate field to the PC or by adding the immediate

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if (Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if (Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

Figure A.27 Typical control flow instructions in RISC-V. All control instructions, except jumps to an address in a register, are PC-relative.

field to the contents of a register. The offset is interpreted as a half word offset for compatibility with the compressed instruction set, R64C, which includes 16-bit instructions.

All branches are conditional. The branch condition is specified by the instruction, and any arithmetic comparison (equal, greater than, less than, and their inverses) is permitted. The branch-target address is specified with a 12-bit signed offset that is shifted left one place (to get 16-bit alignment) and then added to the current program counter. Branches based on the contents of the floating point registers are implemented by executing a floating point comparison (e.g., feq.d or fle.d), which sets an integer register to 0 or 1 based on the comparison, and then executing a beq or bne with x0 as an operand.

The observant reader will have noticed that there are very few 64-bit only instructions in RV64G. Primarily, these are the 64-bit loads and stores and versions of 32-bit, 16-bit, and 8-bit loads that do not sign extend (the default is to sign-extend). To support 32-bit modular arithmetic without additional instructions, there are versions of the instructions that ignore the upper 32 bits of a 64-bit register, such as add and subtract word (addw, subw). Amazingly, everything else just works.

RISC-V Floating-Point Operations

Floating-point instructions manipulate the floating-point registers and indicate whether the operation to be performed is single or double precision. The floating-point operations are add, subtract, multiply, divide, square root, as well as fused multiply-add and multiply-subtract. All floating point instructions begin with the letter `f` and use the suffix `d` for double precision and `s` for single precision (e.g., `fadd.d`, `fadd.s`, `fmul.d`, `fmul.s`, `fmadd.d` `fmadd.s`). Floating-point compares set an integer register based on the comparison, similarly to the integer instruction set-less-than and set-great-than.

In addition to floating-point loads and stores (`flw`, `fsw`, `fld`, `fsd`), instructions are provided for converting between different FP precisions, for moving between integer and FP registers (fmv), and for converting between floating point and integer (fcvt, which uses the integer registers for source or destination as appropriate).

Figure A.28 contains a list of nearly all the RV64G instructions and a summary of their meaning.

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement + contents of a GPR*
lb, lbu, sb	Load byte, load byte unsigned, store byte (to/from integer registers)
lh, lhu, sh	Load half word, load half word unsigned, store half word (to/from integer registers)
lw, lwu, sw	Load word, store word (to/from integer registers)
ld, sd	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
add, addi, addw, addiw, sub, subi, subw, subiw	Add and subtract, with both word and immediate versions
slt, sltu, slti, sltiu	set-less-than with signed and unsigned, and immediate
and, or, xor, andi, ori, xori	and, or, xor, both register-register and register-immediate
lui	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
auipc	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions
Control	*Conditional branches and jumps; PC-relative or through register*
beq, bne, blt, bge, bltu, bgeu	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
jal,jalr	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
flw, fld, fsw, fsd	Load, store, word (single precision), doubleword (double precision)
fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
feq, flt, fle	Compare two floating point registers; result is 0 or 1 stored into a GPR
fmv.x.*, fmv.*.x	Move between the FP register abd GPR, "*" is s or d
fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

Figure A.28 A list of the vast majority of instructions in RV64G. This list can also be found on the back inside cover. This table omits system instructions, synchronization and atomic instructions, configuration instructions, instructions to reset and access performance counters, about 10 instructions in total.

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

RISC-V Instruction Set Usage

To give an idea of which instructions are popular, Figure A.29 shows the frequency of instructions and instruction classes for the SPECint2006 programs, using RV32G.

A.10 Fallacies and Pitfalls

Architects have repeatedly tripped on common, but erroneous, beliefs. In this section we look at a few of them.

Pitfall *Designing a "high-level" instruction set feature specifically oriented to supporting a high-level language structure.*

Attempts to incorporate high-level language features in the instruction set have led architects to provide powerful instructions with a wide range of flexibility. However, often these instructions do more work than is required in the frequent case, or they don't exactly match the requirements of some languages. Many such efforts have been aimed at eliminating what in the 1970s was called the *semantic gap*. Although the idea is to supplement the instruction set with additions that bring the hardware up to the level of the language, the additions can generate what Wulf et al. (1981) have called a *semantic clash*:

> … by giving too much semantic content to the instruction, the computer designer made it possible to use the instruction only in limited contexts. [p. 43]

More often the instructions are simply overkill—they are too general for the most frequent case, resulting in unneeded work and a slower instruction. Again, the VAX `CALLS` is a good example. `CALLS` uses a callee save strategy (the registers to be saved are specified by the callee), *but* the saving is done by the call instruction in the caller. The `CALLS` instruction begins with the arguments pushed on the stack, and then takes the following steps:

1. Align the stack if needed.
2. Push the argument count on the stack.
3. Save the registers indicated by the procedure call mask on the stack (as mentioned in Section A.8). The mask is kept in the called procedure's code—this permits the callee to specify the registers to be saved by the caller even with separate compilation.
4. Push the return address on the stack, and then push the top and base of stack pointers (for the activation record).
5. Clear the condition codes, which sets the trap enable to a known state.
6. Push a word for status information and a zero word on the stack.
7. Update the two stack pointers.
8. Branch to the first instruction of the procedure.

The vast majority of calls in real programs do not require this amount of overhead. Most procedures know their argument counts, and a much faster linkage convention can be established using registers to pass arguments rather than the stack in memory. Furthermore, the `CALLS` instruction forces two registers to be used for linkage, while many languages require only one linkage register. Many attempts to support procedure call and activation stack management have failed to be useful, either because they do not match the language needs or because they are too general and hence too expensive to use.

The VAX designers provided a simpler instruction, `JSB`, that is much faster because it only pushes the return PC on the stack and jumps to the procedure. However, most VAX compilers use the more costly `CALLS` instructions. The call instructions were included in the architecture to standardize the procedure linkage convention. Other computers have standardized their calling convention by agreement among compiler writers and without requiring the overhead of a complex, very general procedure call instruction.

Fallacy *There is such a thing as a typical program.*

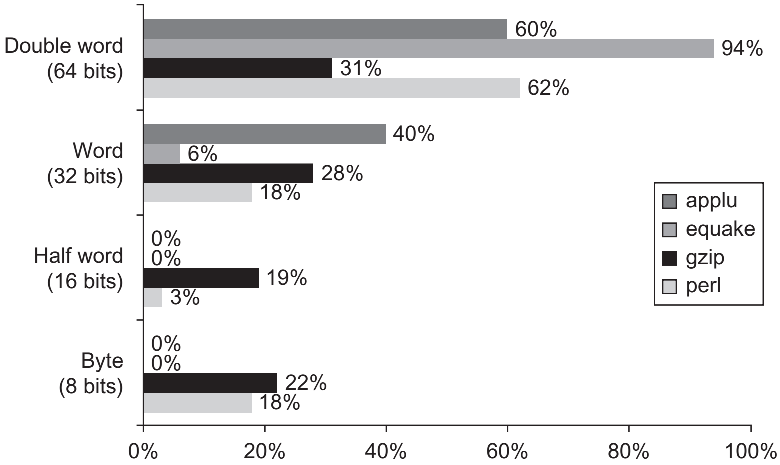


Figure A.30 Data reference size of four programs from SPEC2000. Although you can calculate an average size, it would be hard to claim the average is typical of programs.

Many people would like to believe that there is a single "typical" program that could be used to design an optimal instruction set. For example, see the synthetic benchmarks discussed in Chapter 1. The data in this appendix clearly show that programs can vary significantly in how they use an instruction set. For example, Figure A.30 shows the mix of data transfer sizes for four of the SPEC2000 programs: It would be hard to say what is typical from these four programs. The variations are even larger on an instruction set that supports a class of applications, such as decimal instructions, that are unused by other applications.

Pitfall *Innovating at the instruction set architecture to reduce code size without accounting for the compiler.*

Figure A.31 shows the relative code sizes for four compilers for the MIPS instruction set. Whereas architects struggle to reduce code size by 30%–40%, different compiler strategies can change code size by much larger factors. Similar to performance optimization techniques, the architect should start with the tightest code the compilers can produce before proposing hardware innovations to save space.

Fallacy *An architecture with flaws cannot be successful.*

The 80x86 provides a dramatic example: the instruction set architecture is one only its creators could love (see Appendix K). Succeeding generations of Intel engineers have tried to correct unpopular architectural decisions made in designing the 80x86. For example, the 80x86 supports segmentation, whereas all others picked paging; it uses extended accumulators for integer data, but other processors use general-purpose registers; and it uses a stack for floating-point data, when everyone else abandoned execution stacks long before.

Compiler	**Apogee software version 4.1**	**Green Hills Multi2000 Version 2.0**	**Algorithmics SDE4.0B**	**IDT/c 7.2.1**
Architecture	MIPS IV	MIPS IV	MIPS 32	MIPS 32
Processor	NEC VR5432	NEC VR5000	IDT 32334	IDT 79RC32364
Autocorrelation kernel	1.0	2.1	1.1	2.7
Convolutional encoder kernel	1.0	1.9	1.2	2.4
Fixed-point bit allocation kernel	1.0	2.0	1.2	2.3
Fixed-point complex FFT kernel	1.0	1.1	2.7	1.8
Viterbi GSM decoder kernel	1.0	1.7	0.8	1.1
Geometric mean of five kernels	1.0	1.7	1.4	2.0

Figure A.31 Code size relative to Apogee Software Version 4.1 C compiler for Telecom application of EEMBC benchmarks. The instruction set architectures are virtually identical, yet the code sizes vary by factors of 2. These results were reported February–June 2000.

Despite these major difficulties, the 80x86 architecture has been enormously successful. The reasons are threefold: first, its selection as the microprocessor in the initial IBM PC makes 80x86 binary compatibility extremely valuable. Second, Moore's Law provided sufficient resources for 80x86 microprocessors to translate to an internal RISC instruction set and then execute RISC-like instructions. This mix enables binary compatibility with the valuable PC software base and performance on par with RISC processors. Third, the very high volumes of PC microprocessors mean Intel can easily pay for the increased design cost of hardware translation. In addition, the high volumes allow the manufacturer to go up the learning curve, which lowers the cost of the product.

The larger die size and increased power for translation may be a liability for embedded applications, but it makes tremendous economic sense for the desktop. And its cost-performance in the desktop also makes it attractive for servers, with its main weakness for servers being 32-bit addresses, which was resolved with a 64-bit address extension.

Fallacy *You can design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at the time they were made look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code size efficiency, underestimating how important ease of decoding and pipelining would be five years later. An example in the RISC camp is delayed branch (see Appendix K). It was a simple matter to control pipeline hazards with five-stage pipelines, but a challenge for processors with longer pipelines that issue multiple instructions per clock cycle. In addition, almost all architectures eventually succumb to the lack of sufficient address space. This is one reason that RISC-V has planned for the possibility of 128-bit addresses, although it may be decades before such capability is needed.

In general, avoiding such flaws in the long run would probably mean compromising the efficiency of the architecture in the short run, which is dangerous, since a new instruction set architecture must struggle to survive its first few years.

A.11 Concluding Remarks

The earliest architectures were limited in their instruction sets by the hardware technology of that time. As soon as the hardware technology permitted, computer architects began looking for ways to support high-level languages. This search led to three distinct periods of thought about how to support programs efficiently. In the 1960s, stack architectures became popular. They were viewed as being a good match for high-level languages—and they probably were, given the compiler technology of the day. In the 1970s, the main concern of architects was how to reduce software costs. This concern was met primarily by replacing software with hardware, or by providing high-level architectures that could simplify the task of software designers. The result was both the high-level language computer architecture movement and powerful architectures like the VAX, which has a large number of addressing modes, multiple data types, and a highly orthogonal architecture. In the 1980s, more sophisticated compiler technology and a renewed emphasis on processor performance saw a return to simpler architectures, based mainly on the load-store style of computer.

The following instruction set architecture changes occurred in the 1990s:

- *Address size doubles*—The 32-bit address instruction sets for most desktop and server processors were extended to 64-bit addresses, expanding the width of the registers (among other things) to 64 bits. Appendix K gives three examples of architectures that have gone from 32 bits to 64 bits.
- *Optimization of conditional branches via conditional execution*—In Chapter 3, we see that conditional branches can limit the performance of aggressive computer designs. Hence, there was interest in replacing conditional branches with conditional completion of operations, such as conditional move (see Appendix H), which was added to most instruction sets.
- *Optimization of cache performance via prefetch*—Chapter 2 explains the increasing role of memory hierarchy in the performance of computers, with a cache miss on some computers taking as many instruction times as page faults took on earlier computers. Hence, prefetch instructions were added to try to hide the cost of cache misses by prefetching (see Chapter 2).
- *Support for multimedia*—Most desktop and embedded instruction sets were extended with support for multimedia applications.

- *Faster floating-point operations*—Appendix J describes operations added to enhance floating-point performance, such as operations that perform a multiply and an add and paired single execution, which are part of RISC-V.

Between 1970 and 1985 many thought the primary job of the computer architect was the design of instruction sets. As a result, textbooks of that era emphasize instruction set design, much as computer architecture textbooks of the 1950s and 1960s emphasized computer arithmetic. The educated architect was expected to have strong opinions about the strengths and especially the weaknesses of the popular computers. The importance of binary compatibility in quashing innovations in instruction set design was unappreciated by many researchers and textbook writers, giving the impression that many architects would get a chance to design an instruction set.

The definition of computer architecture today has been expanded to include design and evaluation of the full computer system—not just the definition of the instruction set and not just the processor—and hence there are plenty of topics for the architect to study. In fact, the material in this appendix was a central point of the book in its first edition in 1990, but now is included in an appendix primarily as reference material!

Appendix K may satisfy readers interested in instruction set architecture; it describes a variety of instruction sets, which are either important in the marketplace today or historically important, and it compares nine popular load-store computers with RISC-V.

A.12 Historical Perspective and References

Section M.4 (available online) features a discussion on the evolution of instruction sets and includes references for further reading and exploration of related topics.

Exercises by Gregory D. Peterson

A.1 [10] < A.9 > Compute the effective CPI for an implementation of an embedded RISC-V CPU using Figure A.29. Assume we have made the following measurements of average CPI for each of the instruction types:

Instruction	**Clock cycles**
All ALU operations	1.0
Loads	5.0
Stores	3.0
Branches	
Taken	5.0
Not taken	3.0
Jumps	3.0

Average the instruction frequencies of astar and gcc to obtain the instruction mix.

A.2 [10] < A.9 > Compute the effective CPI for RISC-V using Figure A.29 and the table above. Average the instruction frequencies of bzip and hmmer to obtain the instruction mix. You may assume that all other instructions (for instructions not accounted for by the types in Table A.29) require 3.0 clock cycles each.

A.3 [10] < A.9 > Compute the effective CPI for an implementation of a RISC-V CPU using Figure A.29. Assume we have made the following measurements of average CPI for each of the instruction types:

Instruction	**Clock cycles**
All ALU operations	1.0
Loads	3.5
Stores	2.8
Branches	
Taken	4.0
Not taken	2.0
Jumps	2.4

Average the instruction frequencies of gobmk and mcf to obtain the instruction mix. You may assume that all other instructions (for instructions not accounted for by the types in Table A.29) require 3.0 clock cycles each.

A.4 [10] < A.9 > Compute the effective CPI for RISC-V using Figure A.29 and the table above. Average the instruction frequencies of perlbench and sjeng to obtain the instruction mix.

A.5 [10] < A.8 > Consider this high-level code sequence of three statements:

```
A = B + C;
B = A + C;
D = A - B;
```

Use the technique of copy propagation (see Figure A.20) to transform the code sequence to the point where no operand is a computed value. Note the instances in which the transformation has reduced the computational work of a statement and those cases where the work has increased. What does this suggest about the technical challenge faced in trying to satisfy the desire for optimizing compilers?

A.6 [30] < A.8 > Compiler optimizations may result in improvements to code size and/or performance. Consider one or more of the benchmark programs from the SPEC CPU2017 or the EEMBC benchmark suites. Use the RISC-V processor or a processor available to you along with the GNU C compiler to optimize the benchmark program(s) using no optimization, –O1, –O2, and –O3. Compare the performance and size of the resulting programs. Also compare your results to Figure A.21.

A.7 [20/20/20/25/10] < A.2, A.9 > Consider the following fragment of C code:

```
for (i=0; i<100; i++) {
      A[i]=B[i]+C;
}
```

Assume that `A` and `B` are arrays of 64-bit integers, and `C` and `i` are 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 1000, 3000, 5000, and 7000 for `A`, `B`, `C`, and `i`, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop. Assume all addresses and words are 64 bits.

a. [20] < A.2, A.9 > Write the code for RISC-V. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

b. [20] < A.2 > Write the code for x86. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

c. [20] < A.2 > Write the code for a stack machine. Assume all operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low in cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references. How many instructions are required dynamically? How many memory-data references will be executed?

d. [25] < A.2, A.9 > Instead of the code fragment above, write a routine for computing a matrix multiplication for dense, single precision matrices, also known as SGEMM. For input matrices of size 100 × 100, how many instructions are required dynamically? How many memory-data references will be executed?

e. [10] < A.2, A.9 > As the matrix size increases, how does this affect the number of instructions executed dynamically or the number of memory-data references?

A.8 [25/25] < A.2, A.8, A.9 > Consider the following fragment of C code:

```
for(p=0; p<8; p++) {
    Y[p]=(9798*R[p]+19235*G[p]+3736*B[p])/32768;
    U[p]=(-4784*R[p] – 9437*G[p]+4221*B[p])/32768+128;
    V[p]=(20218*R[p]–16941*G[p]–3277*B[p])/32768+128;
}
```

Assume that `R`, `G`, `B`, `Y`, `U`, and `V` are arrays of 64-bit integers. Assume that all data values and their addresses are kept in memory (at addresses 1000, 2000, 3000, 4000, 5000, and 6000 for `R`, `G`, `B`, `Y`, `U`, and `V`, respectively) except when they are operated on. Assume that values in registers are lost between iterations of the loop. Assume all addresses and words are 64 bits.

a. [25] < A.2, A.9 > Write the code for RISC-V. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes?

b. [25] < A.2 > Write the code for x86. How many instructions are required dynamically? How many memory-data references will be executed? What is the code size in bytes? Compare your results to the multimedia instructions (MMX) and vector implementations discussed in the A.8.

A.9 [10/10/10/10] < A.2, A.7 > For the following, we consider instruction encoding for instruction set architectures.

a. [10] < A.2, A.7 > Consider the case of a processor with an instruction length of 14 bits and with 64 general-purpose registers so the size of the address fields is 6 bits. Is it possible to have instruction encodings for the following?
   - 3 two-address instructions
   - 63 one-address instructions
   - 45 zero-address instructions

b. [10] < A.2, A.7 > Assuming the same instruction length and address field sizes as above, determine if it is possible to have
   - 3 two-address instructions
   - 65 one-address instructions
   - 35 zero-address instructions

   Explain your answer.

c. [10] < A.2, A.7 > Assume the same instruction length and address field sizes as above. Further assume there are already 3 two-address and 24 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?

d. [10] < A.2, A.7 > Assume the same instruction length and address field sizes as above. Further assume there are already 3 two-address and 65 zero-address instructions. What is the maximum number of one-address instructions that can be encoded for this processor?

A.10 [10/15] < A.2 > For the following assume that integer values `A`, `B`, `C`, `D`, `E`, and `F` reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

a. [10] < A.2 > For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute `C = A + B`, and what is the total code size?

b. [15] < A.2 > Some of the instruction set architectures in Figure A.2 destroy operands in the course of computation. This loss of data values from processor internal storage has performance consequences. For each architecture in Figure A.2, write the code sequence to compute:

```
C = A + B
D = A – E
F = C + D
```

In your code, mark each operand that is destroyed during execution and mark each "overhead" instruction that is included just to overcome this loss of data from processor internal storage. What is the total code size, the number of bytes of instructions and data moved to or from memory, the number of overhead instructions, and the number of overhead data bytes for each of your code sequences?

A.11 [15] < A.2, A.7, A.9 > The design of RISC-V provides for 32 general-purpose registers and 32 floating-point registers. If registers are good, are more registers better? List and discuss as many trade-offs as you can that should be considered by instruction set architecture designers examining whether to, and how much to, increase the number of RISC-V registers.

A.12 [5] < A.3 > Consider a C struct that includes the following members:

```
struct foo {
    char a;
    bool b;
    int c;
    double d;
    short e;
    float f;
    double g;
    char *cptr;
    float *fptr;
    int x;
};
```

Note that for C, the compiler must keep the elements of the struct in the same order as given in the struct definition. For a 32-bit machine, what is the size of the foo struct? What is the minimum size required for this struct, assuming you may arrange the order of the struct members as you wish? What about for a 64-bit machine?

A.13 [30] < A.7 > Many computer manufacturers now include tools or simulators that allow you to measure the instruction set usage of a user program. Among the methods in use are machine simulation, hardware-supported trapping, and techniques that instrument the object code module by inserting counters in software or using built-in hardware counters. Pick a processor and tools to instrument user programs. (The open source RISC-V architecture supports a collection of tools. Tools such as the Performance API (PAPI) work with x86 processors.) Use the processor and tools to measure the instruction set mix for one of the SPEC CPU2017 benchmarks. Compare the results to those shown in this chapter.

A.14 [30] < A.8 > Newer processors such as Intel's i7 Kaby Lake include support for AVX2 vector/multimedia instructions. Write a dense matrix multiply function using single-precision values and compile it with different compilers and optimization flags. Linear algebra codes using Basic Linear Algebra Subroutine (BLAS) routines such as SGEMM include optimized versions of dense matrix multiply. Compare the code size and performance of your code to that of BLAS SGEMM. Explore what happens when using double-precision values and DGEMM.

A.15 [30] < A.8 > For the SGEMM code developed above for the i7 processor, include the use of AVX2 intrinsics to improve the performance. In particular, try to vectorize your code to better utilize the AVX hardware. Compare the code size and performance to the original code. Compare your results to Intel's Math Kernel Library (MKL) implementation for SGEMM.

A.16 [30] < A.7, A.9 > The RISC-V processor is open source and boasts an impressive collection of implementations, simulators, compilers, and other tools. See riscv.org for an overview of tools, including spike, a simulator for RISC-V processors. Use spike or another simulator to measure the instruction set mix for some SPEC CPU2017 benchmark programs.

A.17 [35/35/35/35] < A.2–A.8 > gcc targets most modern instruction set architectures (see www.gnu.org/software/gcc/). Create a version of gcc for several architectures that you have access to, such as x86, RISC-V, PowerPC, and ARM.

a. [35] < A.2–A.8 > Compile a subset of SPEC CPU2017 integer benchmarks and create a table of code sizes. Which architecture is best for each program?

b. [35] < A.2–A.8 > Compile a subset of SPEC CPU2017 floating-point benchmarks and create a table of code sizes. Which architecture is best for each program?

c. [35] < A.2–A.8 > Compile a subset of EEMBC AutoBench benchmarks (see www.eembc.org/home.php) and create a table of code sizes. Which architecture is best for each program?

d. [35] < A.2–A.8 > Compile a subset of EEMBC FPBench floating-point benchmarks and create a table of code sizes. Which architecture is best for each program?

A.18 [40] < A.2–A.8 > Power efficiency has become very important for modern processors, particularly for embedded systems. Create a version of gcc for two architectures that you have access to, such as x86, RISC-V, PowerPC, Atom, and ARM. (Note that the different versions of RISC-V can also be explored and compared.) Compile a subset of EEMBC benchmarks while using EnergyBench to measure energy usage during execution. Compare code size, performance, and energy usage for the processors. Which is best for each program?

A.19 [20/15/15/20] Your task is to compare the memory efficiency of four different styles of instruction set architectures. The architecture styles are:

- *Accumulator*—All operations occur between a single register and a memory location.
- *Memory-memory*—All instruction addresses reference only memory locations.
- *Stack*—All operations occur on top of the stack. Push and pop are the only instructions that access memory; all others remove their operands from the stack and replace them with the result. The implementation uses a hardwired stack for only the top two stack entries, which keeps the processor circuit very small and low in cost. Additional stack positions are kept in memory locations, and accesses to these stack positions require memory references.

- *Load-store*—All operations occur in registers, and register-to-register instructions have three register names per instruction.

To measure memory efficiency, make the following assumptions about all four instruction sets:

- All instructions are an integral number of bytes in length.
- The opcode is always one byte (8 bits).
- Memory accesses use direct, or absolute, addressing.
- The variables `A`, `B`, `C`, and `D` are initially in memory.

a. [20] <A.2, A.3> Invent your own assembly language mnemonics (Figure A.2 provides a useful sample to generalize), and for each architecture write the best equivalent assembly language code for this high-level language code sequence:

```
A = B + C;
B = A + C;
D = A – B;
```

b. [15] <A.3> Label each instance in your assembly codes for part (a) where a value is loaded from memory after having been loaded once. Also label each instance in your code where the result of one instruction is passed to another instruction as an operand, and further classify these events as involving storage within the processor or storage in memory.

c. [15] <A.7> Assume that the given code sequence is from a small, embedded computer application that uses a 16-bit memory address and data operands. If a load-store architecture is used, assume it has 16 general-purpose registers. For each architecture answer the following questions: How many instruction bytes are fetched? How many bytes of data are transferred from/to memory? Which architecture is most efficient as measured by total memory traffic (code + data)?

d. [20] <A.7> Now assume a processor with 64-bit memory addresses and data operands. For each architecture answer the questions of part (c). How have the relative merits of the architectures changed for the chosen metrics?

A.20 [30] <A.2, A.3> Use the four different instruction set architecture styles from above, but assume that the memory operations supported include register indirect as well as direct addressing. Invent your own assembly language mnemonics (Figure A.2 provides a useful sample to generalize), and for each architecture, write the best equivalent assembly language code for this fragment of C code:

```
for (i = 0; i <= 100; i++) {
    A[i] = B[i] * C + D ;
}
```

Assume that `A` and `B` are arrays of 64-bit integers, and `C`, `D`, and `i` are 64-bit integers.

A.21 [20/20] < A.3, A.6, A.9 > The size of displacement values needed for the displacement addressing mode or for PC-relative addressing can be extracted from compiled applications. Use a disassembler with one or more of the SPEC CPU2017 or EEMBC benchmarks compiled for the RISC-V processor.

a. [20] < A.3, A.9 > For each instruction using displacement addressing, record the displacement value used. Create a histogram of displacement values. Compare the results to those shown in this appendix in Figure A.8.

b. [20] < A.6, A.9 > For each branch instruction using PC-relative addressing, record the offset value used. Create a histogram of offset values. Compare the results to those shown in this chapter in Figure A.15.

A.22 [15/15/10/10/10/10] < A.3 > The value represented by the hexadecimal number 5249 5343 5643 5055 is to be stored in an aligned 64-bit double word.

a. [15] < A.3 > Using the physical arrangement of the first row in Figure A.5, write the value to be stored using Big Endian byte order. Next, interpret each byte as an ASCII character and below each byte write the corresponding character, forming the character string as it would be stored in Big Endian order.

b. [15] < A.3 > Using the same physical arrangement as in part (a), write the value to be stored using Little Endian byte order, and below each byte write the corresponding ASCII character.

c. [10] < A.3 > What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

d. [10] < A.3 > What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Big Endian byte order?

e. [10] < A.3 > What are the hexadecimal values of all misaligned 2-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

f. [10] < A.3 > What are the hexadecimal values of all misaligned 4-byte words that can be read from the given 64-bit double word when stored in Little Endian byte order?

A.23 [25,25] < A.3, A.9 > The relative frequency of different addressing modes impacts the choices of addressing modes support for an instruction set architecture. Figure A.7 illustrates the relative frequency of addressing modes for three applications on the VAX.

a. [25] < A.3 > Compile one or more programs from the SPEC CPU2017 or EEMBC benchmark suites to target the x86 architecture. Using a disassembler, inspect the instructions and the relative frequency of various addressing modes. Create a histogram to illustrate the relative frequency of the addressing modes. How do your results compare to Figure A.7?

b. [25] < A.3, A.9 > Compile one or more programs from the SPEC CPU2017 or EEMBC benchmark suites to target the RISC-V architecture. Using a disassembler, inspect the instructions and the relative frequency of various addressing modes. Create a histogram to illustrate the relative frequency of the addressing modes. How do your results compare to Figure A.7?

A.24 [Discussion] < A.2–A.12 > Consider typical applications for desktop, server, cloud, and embedded computing. How would instruction set architecture be impacted for machines targeting each of these markets?

B.1 Introduction B-2
B.2 Cache Performance B-15
B.3 Six Basic Cache Optimizations B-22
B.4 Virtual Memory B-40
B.5 Protection and Examples of Virtual Memory B-49
B.6 Fallacies and Pitfalls B-57
B.7 Concluding Remarks B-59
B.8 Historical Perspective and References B-59
Exercises by Amr Zaky B-60

B

Review of Memory Hierarchy

Cache: a safe place for hiding or storing things.

Webster's New World Dictionary of
the American Language,
Second College Edition (1976)

B.1 Introduction

This appendix is a quick refresher of the memory hierarchy, including the basics of cache and virtual memory, performance equations, and simple optimizations. This first section reviews the following 36 terms:

cache
virtual memory
memory stall cycles
direct mapped
valid bit
block address
write through
instruction cache
average memory access time
cache hit
page
miss penalty
fully associative
dirty bit
block offset
write back
data cache
hit time
cache miss
page fault
miss rate
n-way set associative
least recently used
tag field
write allocate
unified cache
misses per instruction
block
locality
address trace
set
random replacement
index field
no-write allocate
write buffer
write stall

If this review goes too quickly, you might want to look at Chapter 7 in *Computer Organization and Design*, which we wrote for readers with less experience.

Cache is the name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor. Because the principle of locality applies at many levels, and taking advantage of locality to improve performance is popular, the term *cache* is now applied whenever buffering is employed to reuse commonly occurring items. Examples include *file caches*, *name caches*, and so on.

When the processor finds a requested data item in the cache, it is called a *cache hit*. When the processor does not find a data item it needs in the cache, a *cache miss* occurs. A fixed-size collection of data containing the requested word, called a *block* or line run, is retrieved from the main memory and placed into the cache. *Temporal locality* tells us that we are likely to need this word again in the near future, so it is useful to place it in the cache where it can be accessed quickly. Because of *spatial locality*, there is a high probability that the other data in the block will be needed soon.

The time required for the cache miss depends on both the latency and bandwidth of the memory. Latency determines the time to retrieve the first word of the block, and bandwidth determines the time to retrieve the rest of this block. A cache miss is handled by hardware and causes processors using in-order execution to pause, or stall, until the data are available. With out-of-order execution, an instruction using the result must still wait, but other instructions may proceed during the miss.

Similarly, not all objects referenced by a program need to reside in main memory. *Virtual memory* means some objects may reside on disk. The address space is

Level	**1**	**2**	**3**	**4**
Name	Registers	Cache	Main memory	Disk storage
Typical size	<4 KiB	32 KiB to 8 MiB	<1 TB	>1 TB
Implementation technology	Custom memory with multiple ports, CMOS	On-chip CMOS SRAM	CMOS DRAM	Magnetic disk or FLASH
Access time (ns)	0.1–0.2	0.5–10	30–150	5,000,000
Bandwidth (MiB/sec)	1,000,000–10,000,000	20,000–50,000	10,000–30,000	100–1000
Managed by	Compiler	Hardware	Operating system	Operating system
Backed by	Cache	Main memory	Disk or FLASH	Other disks and DVD

Figure B.1 The typical levels in the hierarchy slow down and get larger as we move away from the processor for a large workstation or small server. Embedded computers might have no disk storage and much smaller memories and caches. Increasingly, FLASH is replacing magnetic disks, at least for first level file storage. The access times increase as we move to lower levels of the hierarchy, which makes it feasible to manage the transfer less responsively. The implementation technology shows the typical technology used for these functions. The access time is given in nanoseconds for typical values in 2017; these times will decrease over time. Bandwidth is given in megabytes per second between levels in the memory hierarchy. Bandwidth for disk/FLASH storage includes both the media and the buffered interfaces.

usually broken into fixed-size blocks, called *pages*. At any time, each page resides either in main memory or on disk. When the processor references an item within a page that is not present in the cache or main memory, a *palt* occurs, and the entire page is moved from the disk to main memory. Because page faults take so long, they are handled in software and the processor is not stalled. The processor usually switches to some other task while the disk access occurs. From a high-level perspective, the reliance on locality of references and the relative relationships in size and relative cost per bit of cache versus main memory are similar to those of main memory versus disk.

Figure B.1 shows the range of sizes and access times of each level in the memory hierarchy for computers ranging from high-end desktops to low-end servers.

Cache Performance Review

Because of locality and the higher speed of smaller memories, a memory hierarchy can substantially improve performance. One method to evaluate cache performance is to expand our processor execution time equation from Chapter 1. We now account for the number of cycles during which the processor is stalled waiting for a memory access, which we call the *memory stall cycles*. The performance is then the product of the clock cycle time and the sum of the processor cycles and the memory stall cycles:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

This equation assumes that the CPU clock cycles include the time to handle a cache hit and that the processor is stalled during a cache miss. Section B.2 reexamines this simplifying assumption.

The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the *miss penalty:*

$$\begin{aligned}\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\end{aligned}$$

The advantage of the last form is that the components can be easily measured. We already know how to measure instruction count (IC). (For speculative processors, we only count instructions that commit.) Measuring the number of memory references per instruction can be done in the same fashion; every instruction requires an instruction access, and it is easy to decide if it also requires a data access.

Note that we calculated miss penalty as an average, but we will use it herein as if it were a constant. The memory behind the cache may be busy at the time of the miss because of prior memory requests or memory refresh. The number of clock cycles also varies at interfaces between different clocks of the processor, bus, and memory. Thus, please remember that using a single number for miss penalty is a simplification.

The component *miss rate* is simply the fraction of cache accesses that result in a miss (i.e., number of accesses that miss divided by number of accesses). Miss rates can be measured with cache simulators that take an *address trace* of the instruction and data references, simulate the cache behavior to determine which references hit and which miss, and then report the hit and miss totals. Many microprocessors today provide hardware to count the number of misses and memory references, which is a much easier and faster way to measure miss rate.

The preceding formula is an approximation because the miss rates and miss penalties are often different for reads and writes. Memory stall clock cycles could then be defined in terms of the number of memory accesses per instruction, miss penalty (in clock cycles) for reads and writes, and miss rate for reads and writes:

$$\begin{aligned}\text{Memory stall clock cycles} = {} & \text{IC} \times \text{Reads per instruction} \times \text{Read miss rate} \times \text{Read miss penalty}\\ & + \text{IC} \times \text{Writes per instruction} \times \text{Write miss rate} \times \text{Write miss penalty}\end{aligned}$$

We usually simplify the complete formula by combining the reads and writes and finding the average miss rates and miss penalty for reads *and* writes:

$$\text{Memory stall clock cycles} = \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}$$

The miss rate is one of the most important measures of cache design, but, as we will see in later sections, not the only measure.

Example Assume we have a computer where the cycles per instruction (CPI) is 1.0 when all memory accesses hit in the cache. The only data accesses are loads and stores, and these total 50% of the instructions. If the miss penalty is 50 clock cycles and the miss rate is 1%, how much faster would the computer be if all instructions were cache hits?

Answer First compute the performance for the computer that always hits:

$$\begin{aligned}\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}\\ &= (\text{IC} \times \text{CPI} + 0) \times \text{Clock cycle}\\ &= \text{IC} \times 1.0 \times \text{Clock cycle}\end{aligned}$$

Now for the computer with the real cache, first we compute memory stall cycles:

$$\begin{aligned}\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss penalty}\\ &= \text{IC} \times (1 + 0.5) \times 0.01 \times 50\\ &= \text{IC} \times 0.75\end{aligned}$$

where the middle term $(1+0.5)$ represents one instruction access and 0.5 data accesses per instruction. The total performance is thus

$$\begin{aligned}\text{CPU execution time}_{\text{cache}} &= (\text{IC} \times 1.0 + \text{IC} \times 0.75) \times \text{Clock cycle}\\ &= 1.75 \times \text{IC} \times \text{Clock cycle}\end{aligned}$$

The performance ratio is the inverse of the execution times:

$$\begin{aligned}\frac{\text{CPU execution time}_{\text{cache}}}{\text{CPU execution time}} &= \frac{1.75 \times \text{IC} \times \text{Clock cycle}}{1.0 \times \text{IC} \times \text{Clock cycle}}\\ &= 1.75\end{aligned}$$

The computer with no cache misses is 1.75 times faster.

Some designers prefer measuring miss rate as *misses per instruction* rather than misses per memory reference. These two are related:

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate} \times \text{Memory accesses}}{\text{Instruction count}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

The latter formula is useful when you know the average number of memory accesses per instruction because it allows you to convert miss rate into misses per instruction, and vice versa. For example, we can turn the miss rate per memory reference in the previous example into misses per instruction:

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} = 0.02 \times (1.5) = 0.030$$

By the way, misses per instruction are often reported as misses per 1000 instructions to show integers instead of fractions. Thus, the preceding answer could also be expressed as 30 misses per 1000 instructions.

The advantage of misses per instruction is that it is independent of the hardware implementation. For example, speculative processors fetch about twice as many instructions as are actually committed, which can artificially reduce the miss rate if measured as misses per memory reference rather than per instruction. The drawback is that misses per instruction is architecture dependent; for example, the average number of memory accesses per instruction may be very different for an 80x86 versus RISC V. Thus, misses per instruction are most popular with architects working with a single computer family, although the similarity of RISC architectures allows one to give insights into others.

Example To show equivalency between the two miss rate equations, let's redo the preceding example, this time assuming a miss rate per 1000 instructions of 30. What is memory stall time in terms of instruction count?

Answer Recomputing the memory stall cycles:

$$\begin{aligned}
\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty} \\
&= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \\
&= \text{IC}/1000 \times \frac{\text{Misses}}{\text{Intruction} \times 1000} \times \text{Miss penalty} \\
&= \text{IC}/1000 \times 30 \times 25 \\
&= \text{IC}/1000 \times 750 \\
&= \text{IC} \times 0.75
\end{aligned}$$

We get the same answer as on page B-5, showing equivalence of the two equations.

Four Memory Hierarchy Questions

We continue our introduction to caches by answering the four common questions for the first level of the memory hierarchy:

Q1: Where can a block be placed in the upper level? (*block placement*)

Q2: How is a block found if it is in the upper level? (*block identification*)

Q3: Which block should be replaced on a miss? (*block replacement*)

Q4: What happens on a write? (*write strategy*)

The answers to these questions help us understand the different trade-offs of memories at different levels of a hierarchy; hence, we ask these four questions on every example.

Q1: Where Can a Block be Placed in a Cache?

Figure B.2 shows that the restrictions on where a block is placed create three categories of cache organization:

- If each block has only one place it can appear in the cache, the cache is said to be *direct mapped*. The mapping is usually

  (*Block address*) MOD (*Number of blocks in cache*)

- If a block can be placed anywhere in the cache, the cache is said to be *fully associative*.

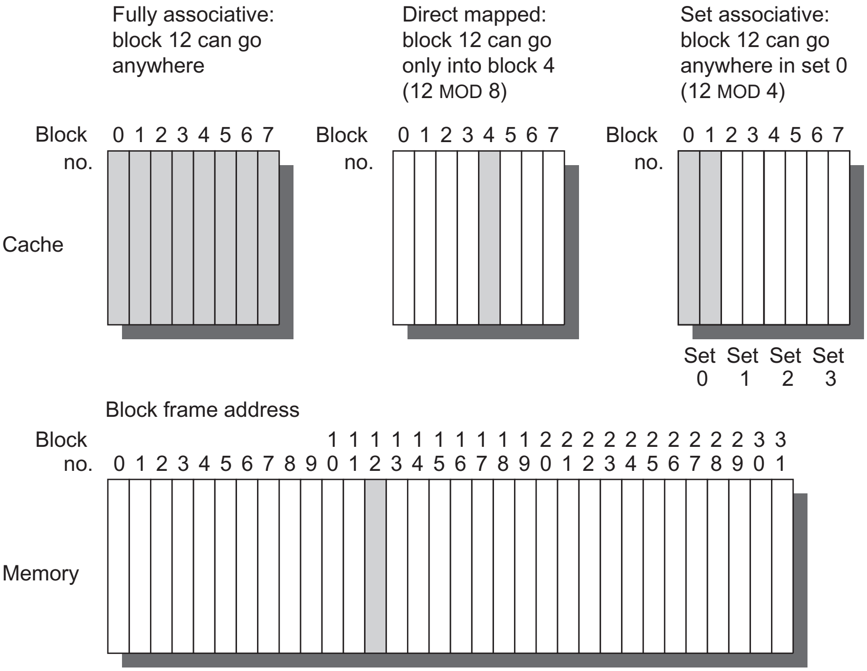

Fully associative:
block 12 can go
anywhere
Direct mapped:
block 12 can go
only into block 4
(12 MOD 8)
Set associative:
block 12 can go
anywhere in set 0
(12 MOD 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory

Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called *two-way set associative*. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

- If a block can be placed in a restricted set of places in the cache, the cache is *set associative*. A *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by *bit selection*; that is,

  (*Block address*) MOD (*Number of sets in cache*)

  If there are *n* blocks in a set, the cache placement is called *n-way set associative*.

The range of caches from direct mapped to fully associative is really a continuum of levels of set associativity. Direct mapped is simply one-way set associative, and a fully associative cache with *m* blocks could be called "*m*-way set associative." Equivalently, direct mapped can be thought of as having *m* sets, and fully associative as having one set.

The vast majority of processor caches today are direct mapped, two-way set associative, or four-way set associative, for reasons we will see shortly.

Q2: How Is a Block Found If It Is in the Cache?

Caches have an address tag on each block frame that gives the block address. The tag of every cache block that might contain the desired information is checked to see if it matches the block address from the processor. As a rule, all possible tags are searched in parallel because speed is critical.

There must be a way to know that a cache block does not have valid information. The most common procedure is to add a *valid bit* to the tag to say whether or not this entry contains a valid address. If the bit is not set, there cannot be a match on this address.

Before proceeding to the next question, let's explore the relationship of a processor address to the cache. Figure B.3 shows how an address is divided. The first division is between the *block address* and the *block offset.* The block frame address can be further divided into the *tag field* and the *index field.* The block offset field selects the desired data from the block, the index field selects the set, and the tag field is compared against it for a hit. Although the comparison could be made on more of the address than the tag, there is no need because of the following:

- The offset should not be used in the comparison, because the entire block is present or not, and hence all block offsets result in a match by definition.

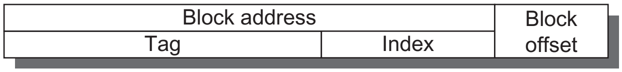


Figure B.3 The three portions of an address in a set associative or direct-mapped cache. The tag is used to check all the blocks in the set, and the index is used to select the set. The block offset is the address of the desired data within the block. Fully associative caches have no index field.

- Checking the index is redundant, because it was used to select the set to be checked. An address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0; set 1 must have an index value of 1; and so on. This optimization saves hardware and power by reducing the width of memory size for the cache tag.

If the total cache size is kept the same, increasing associativity increases the number of blocks per set, thereby decreasing the size of the index and increasing the size of the tag. That is, the tag-index boundary in Figure B.3 moves to the right with increasing associativity, with the end point of fully associative caches having no index field.

Q3: Which Block Should be Replaced on a Cache Miss?

When a miss occurs, the cache controller must select a block to be replaced with the desired data. A benefit of direct-mapped placement is that hardware decisions are simplified—in fact, so simple that there is no choice: only one block frame is checked for a hit, and only that block can be replaced. With fully associative or set associative placement, there are many blocks to choose from on a miss. There are three primary strategies employed for selecting which block to replace:

- *Random*—To spread allocation uniformly, candidate blocks are randomly selected. Some systems generate pseudorandom block numbers to get reproducible behavior, which is particularly useful when debugging hardware.
- *Least recently used* (LRU)—To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. Relying on the past to predict the future, the block replaced is the one that has been unused for the longest time. LRU relies on a corollary of locality: if recently used blocks are likely to be used again, then a good candidate for disposal is the least recently used block.
- *First in, first out* (FIFO)—Because LRU can be complicated to calculate, this approximates LRU by determining the *oldest* block rather than the LRU.

A virtue of random replacement is that it is simple to build in hardware. As the number of blocks to keep track of increases, LRU becomes increasingly expensive and is usually only approximated. A common approximation (often called pseudo-LRU) has a set of bits for each set in the cache with each bit corresponding to a single way (a *way* is bank in a set associative cache; there are four ways in four-way set associative cache) in the cache. When a set is accessed, the bit corresponding to the way containing the desired block is turned on; if all the bits associated with a set are turned on, they are reset with the exception of the most recently turned on bit. When a block must be replaced, the processor chooses a block from the way whose bit is turned off, often randomly if more than one choice is available. This approximates LRU, because the block that is replaced will not have

	Associativity								
	Two-way			Four-way			Eight-way		
Size	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure B.4 Data cache misses per 1000 instructions comparing least recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

been accessed since the last time that all the blocks in the set were accessed. Figure B.4 shows the difference in miss rates between LRU, random, and FIFO replacement.

Q4: What Happens on a Write?

Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions don't write to memory. Figures A.32 and A.33 in Appendix A suggest a mix of 10% stores and 26% loads for RISC V programs, making writes 10%/(100% + 26% + 10%) or about 7% of the overall memory traffic. Of the *data cache* traffic, writes are 10%/(26% + 10%) or about 28%. Making the common case fast means optimizing caches for reads, especially because processors traditionally wait for reads to complete but need not wait for writes. Amdahl's Law (Section 1.9) reminds us, however, that high-performance designs cannot neglect the speed of writes.

Fortunately, the common case is also the easy case to make fast. The block can be read from the cache at the same time that the tag is read and compared, so the block read begins as soon as the block address is available. If the read is a hit, the requested part of the block is passed on to the processor immediately. If it is a miss, there is no benefit—but also no harm except more power in desktop and server computers; just ignore the value read.

Such optimism is not allowed for writes. Modifying a block cannot begin until the tag is checked to see if the address is a hit. Because tag checking cannot occur in parallel, writes usually take longer than reads. Another complexity is that the processor also specifies the size of the write, usually between 1 and 8 bytes; only that portion of a block can be changed. In contrast, reads can access more bytes than necessary without fear.

The write policies often distinguish cache designs. There are two basic options when writing to the cache:

- *Write through*—The information is written to both the block in the cache *and* to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

To reduce the frequency of writing back blocks on replacement, a feature called the *dirty bit* is commonly used. This status bit indicates whether the block is *dirty* (modified while in the cache) or *clean* (not modified). If it is clean, the block is not written back on a miss, because identical information to the cache is found in lower levels.

Both write back and write through have their advantages. With write back, writes occur at the speed of the cache memory, and multiple writes within a block require only one write to the lower-level memory. Because some writes don't go to memory, write back uses less memory bandwidth, making write back attractive in multiprocessors. Since write back uses the rest of the memory hierarchy and memory interconnect less than write through, it also saves power, making it attractive for embedded applications.

Write through is easier to implement than write back. The cache is always clean, so unlike write back read misses never result in writes to the lower level. Write through also has the advantage that the next lower level has the most current copy of the data, which simplifies data coherency. Data coherency is important for multiprocessors and for I/O, which we examine in Chapter 4 and Appendix D. Multilevel caches make write through more viable for the upper-level caches, as the writes need only propagate to the next lower level rather than all the way to main memory.

As we will see, I/O and multiprocessors are fickle: they want write back for processor caches to reduce the memory traffic and write through to keep the cache consistent with lower levels of the memory hierarchy.

When the processor must wait for writes to complete during write through, the processor is said to *write stall.* A common optimization to reduce write stalls is a *write buffer,* which allows the processor to continue as soon as the data are written to the buffer, thereby overlapping processor execution with memory updating. As we will see shortly, write stalls can occur even with write buffers.

Because the data are not needed on a write, there are two options on a write miss:

- *Write allocate*—The block is allocated on a write miss, followed by the preceding write hit actions. In this natural option, write misses act like read misses.
- *No-write allocate*—This apparently unusual alternative is write misses do *not* affect the cache. Instead, the block is modified only in the lower-level memory.

Thus, blocks stay out of the cache in no-write allocate until the program tries to read the blocks, but even blocks that are only written will still be in the cache with write allocate. Let's look at an example.

Example Assume a fully associative write-back cache with many cache entries that starts empty. Following is a sequence of five memory operations (the address is in square brackets):

```
Write Mem[100];
Write Mem[100];
Read  Mem[200];
Write Mem[200];
Write Mem[100].
```

What are the number of hits and misses when using no-write allocate versus write allocate?

Answer For no-write allocate, the address 100 is not in the cache, and there is no allocation on write, so the first two writes will result in misses. Address 200 is also not in the cache, so the read is also a miss. The subsequent write to address 200 is a hit. The last write to 100 is still a miss. The result for no-write allocate is four misses and one hit.

For write allocate, the first accesses to 100 and 200 are misses, and the rest are hits because 100 and 200 are both found in the cache. Thus, the result for write allocate is two misses and three hits.

Either write miss policy could be used with write through or write back. Usually, write-back caches use write allocate, hoping that subsequent writes to that block will be captured by the cache. Write-through caches often use no-write allocate. The reasoning is that even if there are subsequent writes to that block, the writes must still go to the lower-level memory, so what's to be gained?

An Example: The Opteron Data Cache

To give substance to these ideas, Figure B.5 shows the organization of the data cache in the AMD Opteron microprocessor. The cache contains 65,536 (64 K) bytes of data in 64-byte blocks with two-way set associative placement, least-recently used replacement, write back, and write allocate on a write miss.

Let's trace a cache hit through the steps of a hit as labeled in Figure B.5. (The four steps are shown as circled numbers.) As described in Section B.5, the Opteron presents a 48-bit virtual address to the cache for tag comparison, which is simultaneously translated into a 40-bit physical address.

The reason Opteron doesn't use all 64 bits of virtual address is that its designers don't think anyone needs that much virtual address space yet, and the smaller size simplifies the Opteron virtual address mapping. The designers plan to grow the virtual address in future microprocessors.

The physical address coming into the cache is divided into two fields: the 34-bit block address and the 6-bit block offset ($64=2^6$ and $34+6=40$). The block

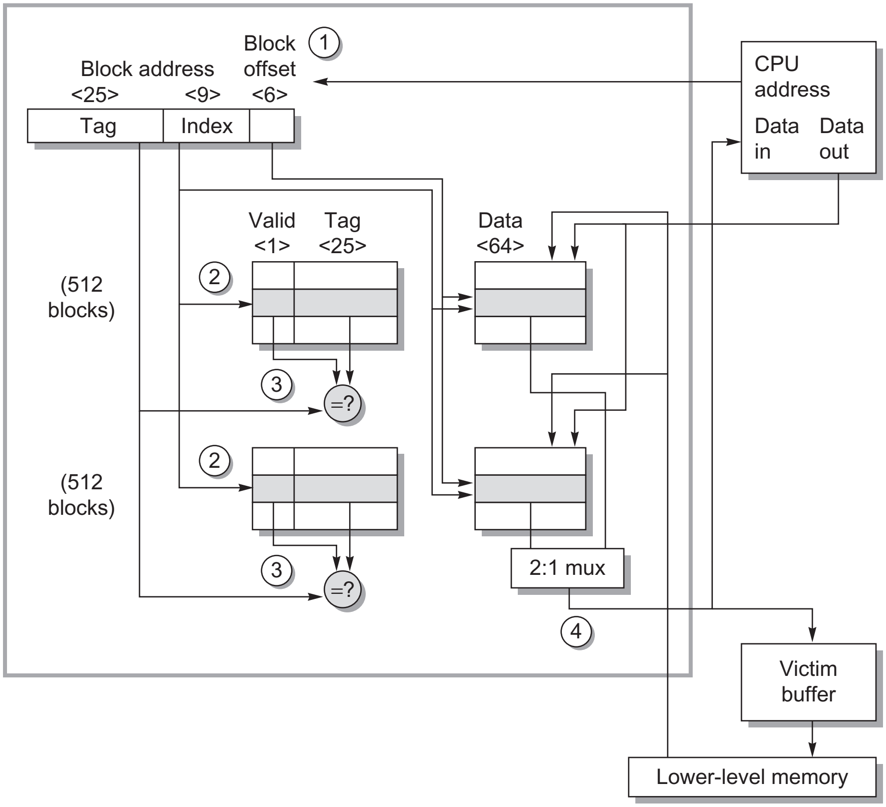

Block address
<25>
<9>
Block offset
<6>
①
Tag
Index
CPU address
Data in
Data out
Valid <1>
Tag <25>
Data <64>
(512 blocks)
(512 blocks)
②
②
③
③
=?
=?
2:1 mux
④
Victim buffer
Lower-level memory

Figure B.5 The organization of the data cache in the Opteron microprocessor. The 64 KiB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache. The size of address leaving the processor is 40 bits because it is a physical address and not a virtual address. Figure B.24 on page B-47 explains how the Opteron maps from virtual to physical for a cache access.

address is further divided into an address tag and cache index. Step 1 shows this division.

The cache index selects the tag to be tested to see if the desired block is in the cache. The size of the index depends on cache size, block size, and set associativity. For the Opteron cache the set associativity is set to two, and we calculate the index as follows:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}} = \frac{65{,}536}{64 \times 2} = 512 = 2^9$$

Hence, the index is 9 bits wide, and the tag is 34 − 9 or 25 bits wide. Although that is the index needed to select the proper block, 64 bytes is much more than the processor wants to consume at once. Hence, it makes more sense to organize the data portion of the cache memory 8 bytes wide, which is the natural data word of the 64-bit Opteron processor. Thus, in addition to 9 bits to index the proper cache block, 3 more bits from the block offset are used to index the proper 8 bytes. Index selection is step 2 in Figure B.5.

After reading the two tags from the cache, they are compared with the tag portion of the block address from the processor. This comparison is step 3 in the figure. To be sure the tag contains valid information, the valid bit must be set or else the results of the comparison are ignored.

Assuming one tag does match, the final step is to signal the processor to load the proper data from the cache by using the winning input from a 2:1 multiplexor. The Opteron allows 2 clock cycles for these four steps, so the instructions in the following 2 clock cycles would wait if they tried to use the result of the load.

Handling writes is more complicated than handling reads in the Opteron, as it is in any cache. If the word to be written is in the cache, the first three steps are the same. Because the Opteron executes out of order, only after it signals that the instruction has committed and the cache tag comparison indicates a hit are the data written to the cache.

So far we have assumed the common case of a cache hit. What happens on a miss? On a read miss, the cache sends a signal to the processor telling it the data are not yet available, and 64 bytes are read from the next level of the hierarchy. The latency is 7 clock cycles to the first 8 bytes of the block, and then 2 clock cycles per 8 bytes for the rest of the block. Because the data cache is set associative, there is a choice on which block to replace. Opteron uses LRU, which selects the block that was referenced longest ago, so every access must update the LRU bit. Replacing a block means updating the data, the address tag, the valid bit, and the LRU bit.

Because the Opteron uses write back, the old data block could have been modified, and hence it cannot simply be discarded. The Opteron keeps 1 dirty bit per block to record if the block was written. If the “victim” was modified, its data and address are sent to the victim buffer. (This structure is similar to a *write buffer* in other computers.) The Opteron has space for eight victim blocks. In parallel with other cache actions, it writes victim blocks to the next level of the hierarchy. If the victim buffer is full, the cache must wait.

A write miss is very similar to a read miss, because the Opteron allocates a block on a read or a write miss.

We have seen how it works, but the *data* cache cannot supply all the memory needs of the processor: the processor also needs instructions. Although a single cache could try to supply both, it can be a bottleneck. For example, when a load or store instruction is executed, the pipelined processor will simultaneously request both a data word *and* an instruction word. Hence, a single cache would present a structural hazard for loads and stores, leading to stalls. One simple way to conquer

Size (KiB)	Instruction cache	Data cache	Unified cache
8	8.16	44.0	63.0
16	3.82	40.9	51.0
32	1.36	38.4	43.3
64	0.61	36.9	39.4
128	0.30	35.3	36.2
256	0.02	32.6	32.9

Figure B.6 Miss per 1000 instructions for instruction, data, and unified caches of different sizes. The percentage of instruction references is about 74%. The data are for two-way associative caches with 64-byte blocks for the same computer and benchmarks as Figure B.4.

this problem is to divide it: one cache is dedicated to instructions and another to data. Separate caches are found in most recent processors, including the Opteron. Hence, it has a 64 KiB instruction cache as well as the 64 KiB data cache.

The processor knows whether it is issuing an instruction address or a data address, so there can be separate ports for both, thereby doubling the bandwidth between the memory hierarchy and the processor. Separate caches also offer the opportunity of optimizing each cache separately: different capacities, block sizes, and associativities may lead to better performance. (In contrast to the instruction caches and data caches of the Opteron, the terms *unified* or *mixed* are applied to caches that can contain either instructions or data.)

Figure B.6 shows that instruction caches have lower miss rates than data caches. Separating instructions and data removes misses due to conflicts between instruction blocks and data blocks, but the split also fixes the cache space devoted to each type. Which is more important to miss rates? A fair comparison of separate instruction and data caches to unified caches requires the total cache size to be the same. For example, a separate 16 KiB instruction cache and 16 KiB data cache should be compared with a 32 KiB unified cache. Calculating the average miss rate with separate instruction and data caches necessitates knowing the percentage of memory references to each cache. From the data in Appendix A we find the split is 100%/(100% + 26% + 10%) or about 74% instruction references to (26% + 10%)/(100% + 26% + 10%) or about 26% data references. Splitting affects performance beyond what is indicated by the change in miss rates, as we will see shortly.

B.2 Cache Performance

Because instruction count is independent of the hardware, it is tempting to evaluate processor performance using that number. Such indirect performance measures have waylaid many a computer designer. The corresponding temptation for evaluating memory hierarchy performance is to concentrate on miss rate because it, too, is independent of the speed of the hardware. As we will see, miss rate can be just as misleading as instruction count. A better measure of memory hierarchy performance is the *average memory access time*:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

where *hit time* is the time to hit in the cache; we have seen the other two terms before. The components of average access time can be measured either in absolute time—say, 0.25–1.0 ns on a hit—or in the number of clock cycles that the processor waits for the memory—such as a miss penalty of 150–200 clock cycles. Remember that average memory access time is still an indirect measure of performance; although it is a better measure than miss rate, it is not a substitute for execution time.

This formula can help us decide between split caches and a unified cache.

Example Which has the lower miss rate: a 16 KiB instruction cache with a 16 KiB data cache or a 32 KiB unified cache? Use the miss rates in Figure B.6 to help calculate the correct answer, assuming 36% of the instructions are data transfer instructions. Assume a hit takes 1 clock cycle and the miss penalty is 100 clock cycles. A load or store hit takes 1 extra clock cycle on a unified cache if there is only one cache port to satisfy two simultaneous requests. Using the pipelining terminology of Chapter 3, the unified cache leads to a structural hazard. What is the average memory access time in each case? Assume write-through caches with a write buffer and ignore stalls due to the write buffer.

Answer First let's convert misses per 1000 instructions into miss rates. Solving the preceding general formula, the miss rate is

$$\text{Miss rate} = \frac{\dfrac{\text{Misses}}{1000\,\text{Instructions}}/1000}{\dfrac{\text{Memory accesses}}{\text{Instruction}}}$$

Because every instruction access has exactly one memory access to fetch the instruction, the instruction miss rate is

$$\text{Miss rate}_{16\,\text{KB instruction}} = \frac{3.82/1000}{1.00} = 0.004$$

Because 36% of the instructions are data transfers, the data miss rate is

$$\text{Miss rate}_{16\,\text{KB data}} = \frac{40.9/1000}{0.36} = 0.114$$

The unified miss rate needs to account for instruction and data accesses:

$$\text{Miss rate}_{32\,\text{KB unified}} = \frac{43.3/1000}{1.00 + 0.36} = 0.0318$$

As stated herein, about 74% of the memory accesses are instruction references. Thus, the overall miss rate for the split caches is

$$(74\% \times 0.004) + (26\% \times 0.114) = 0.0326$$

Thus, a 32 KiB unified cache has a slightly lower effective miss rate than two 16 KiB caches.

The average memory access time formula can be divided into instruction and data accesses:

$$\begin{aligned}&\text{Average memory access time}\\&\quad = \%\,\text{instructions} \times (\text{Hit time} + \text{Instruction miss rate} \times \text{Miss penalty})\\&\qquad + \%\,\text{data} \times (\text{Hit time} + \text{Data miss rate} \times \text{Miss penalty})\end{aligned}$$

Therefore, the time for each organization is

$$\begin{aligned}&\text{Average memory access time}_{\text{split}}\\&\quad = 74\% \times (1 + 0.004 \times 200) + 26\% \times (1 + 0.114 \times 200)\\&\quad = (74\% \times 1.80) + (26\% \times 23.80) = 1.332 + 6.188 = 7.52\\&\text{Average memory access time}_{\text{unified}}\\&\quad = 74\% \times (1 + 0.0318 \times 200) + 26\% \times (1 + 1 + 0.0318 \times 200)\\&\quad = (74\% \times 7.36) + (26\% \times 8.36) = 5.446 + 2.174 = 7.62\end{aligned}$$

Hence, the split caches in this example—which offer two memory ports per clock cycle, thereby avoiding the structural hazard—have a better average memory access time than the single-ported unified cache despite having a worse effective miss rate.

Average Memory Access Time and Processor Performance

An obvious question is whether average memory access time due to cache misses predicts processor performance.

First, there are other reasons for stalls, such as contention due to I/O devices using memory. Designers often assume that all memory stalls are due to cache misses, because the memory hierarchy typically dominates other reasons for stalls. We use this simplifying assumption here, but be sure to account for *all* memory stalls when calculating final performance.

Second, the answer also depends on the processor. If we have an in-order execution processor (see Chapter 3), then the answer is basically yes. The processor stalls during misses, and the memory stall time is strongly correlated to average memory access time. Let's make that assumption for now, but we'll return to out-of-order processors in the next subsection.

As stated in the previous section, we can model CPU time as:

$$\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{Clock cycle time}$$

This formula raises the question of whether the clock cycles for a cache hit should be considered part of CPU execution clock cycles or part of memory stall clock cycles. Although either convention is defensible, the most widely accepted is to include hit clock cycles in CPU execution clock cycles.

We can now explore the impact of caches on performance.

Example Let's use an in-order execution computer for the first example. Assume that the cache miss penalty is 200 clock cycles, and all instructions usually take 1.0 clock cycles (ignoring memory stalls). Assume that the average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. What is the impact on performance when behavior of the cache is included? Calculate the impact using both misses per instruction and miss rate.

Answer

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}$$

The performance, including cache misses, is

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (30/1000 \times 200)] \times \text{Clock cycle time}\\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

Now calculating performance using miss rate:

$$\text{CPU time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}$$

$$\begin{aligned}\text{CPU time}_{\text{with cache}} &= \text{IC} \times [1.0 + (1.5 \times 2\% \times 200)] \times \text{Clock cycle time}\\ &= \text{IC} \times 7.00 \times \text{Clock cycle time}\end{aligned}$$

The clock cycle time and instruction count are the same, with or without a cache. Thus, CPU time increases sevenfold, with CPI from 1.00 for a "perfect cache" to 7.00 with a cache that can miss. Without any memory hierarchy at all the CPI would increase again to $1.0 + 200 \times 1.5$ or 301—a factor of more than 40 times longer than a system with a cache!

As this example illustrates, cache behavior can have enormous impact on performance. Furthermore, cache misses have a double-barreled impact on a processor with a low CPI and a fast clock:

1. The lower the $\text{CPI}_{\text{execution}}$, the higher the *relative* impact of a fixed number of cache miss clock cycles.
2. When calculating CPI, the cache miss penalty is measured in processor clock cycles for a miss. Therefore, even if memory hierarchies for two computers are

identical, the processor with the higher clock rate has a larger number of clock cycles per miss and hence a higher memory portion of CPI.

The importance of the cache for processors with low CPI and high clock rates is thus greater, and, consequently, greater is the danger of neglecting cache behavior in assessing performance of such computers. Amdahl's Law strikes again!

Although minimizing average memory access time is a reasonable goal—and we will use it in much of this appendix—keep in mind that the final goal is to reduce processor execution time. The next example shows how these two can differ.

Example What is the impact of two different cache organizations on the performance of a processor? Assume that the CPI with a perfect cache is 1.0, the clock cycle time is 0.35 ns, there are 1.4 memory references per instruction, the size of both caches is 128 KiB, and both have a block size of 64 bytes. One cache is direct mapped and the other is two-way set associative. Figure B.5 shows that for set associative caches we must add a multiplexor to select between the blocks in the set depending on the tag match. Because the speed of the processor can be tied directly to the speed of a cache hit, assume the processor clock cycle time must be stretched 1.35 times to accommodate the selection multiplexor of the set associative cache. To the first approximation, the cache miss penalty is 65 ns for either cache organization. (In practice, it is normally rounded up or down to an integer number of clock cycles.) First, calculate the average memory access time and then processor performance. Assume the hit time is 1 clock cycle, the miss rate of a direct-mapped 128 KiB cache is 2.1%, and the miss rate for a two-way set associative cache of the same size is 1.9%.

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Thus, the time for each organization is

$$\text{Average memory access time}_{\text{1-way}} = 0.35 + (.021 \times 65) = 1.72\,\text{ns}$$

$$\text{Average memory access time}_{\text{2-way}} = 0.35 \times 1.35 + (.019 \times 65) = 1.71\,\text{ns}$$

The average memory access time is better for the two-way set-associative cache.

The processor performance is

$$\begin{aligned}\text{CPU time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}\\ &= \text{IC} \times \left[(\text{CPI}_{\text{execution}} \times \text{Clock cycle time})\right.\\ &\quad \left. + \left(\text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \times \text{Clock cycle time}\right)\right]\end{aligned}$$

Substituting 65 ns for (Miss penalty × Clock cycle time), the performance of each cache organization is

$$\text{CPU time}_{\text{1-way}} = \text{IC} \times [1.0 \times 0.35 + (0.021 \times 1.4 \times 65)] = 2.26 \times \text{IC}$$

$$\text{CPU time}_{\text{2-way}} = \text{IC} \times [1.0 \times 0.35 \times 1.35 + (0.019 \times 1.4 \times 65)] = 2.20 \times \text{IC}$$

and relative performance is

$$\frac{\text{CPU time}_{\text{2-way}}}{\text{CPU time}_{\text{1-way}}} = \frac{2.26 \times \text{Instruction count}}{2.20 \times \text{Instruction count}} = 1.03$$

In contrast to the results of average memory access time comparison, the direct-mapped cache leads to slightly better average performance because the clock cycle is stretched for *all* instructions for the two-way set associative case, even if there are fewer misses. Because CPU time is our bottom-line evaluation and because direct mapped is simpler to build, the preferred cache is direct mapped in this example.

Miss Penalty and Out-of-Order Execution Processors

For an out-of-order execution processor, how do you define "miss penalty"? Is it the full latency of the miss to memory, or is it just the "exposed" or nonoverlapped latency when the processor must stall? This question does not arise in processors that stall until the data miss completes.

Let's redefine memory stalls to lead to a new definition of miss penalty as nonoverlapped latency:

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

Similarly, as some out-of-order processors stretch the hit time, that portion of the performance equation could be divided by total hit latency less overlapped hit latency. This equation could be further expanded to account for contention for memory resources in an out-of-order processor by dividing total miss latency into latency without contention and latency due to contention. Let's just concentrate on miss latency.

We now have to decide the following:

- *Length of memory latency*—What to consider as the start and the end of a memory operation in an out-of-order processor.
- *Length of latency overlap*—What is the start of overlap with the processor (or, equivalently, when do we say a memory operation is stalling the processor)?

Given the complexity of out-of-order execution processors, there is no single correct definition.

Because only committed operations are seen at the retirement pipeline stage, we say a processor is stalled in a clock cycle if it does not retire the maximum possible number of instructions in that cycle. We attribute that stall to the first instruction that could not be retired. This definition is by no means foolproof. For example, applying an optimization to improve a certain stall time may not always improve execution time because another type of stall—hidden behind the targeted stall—may now be exposed.

For latency, we could start measuring from the time the memory instruction is queued in the instruction window, or when the address is generated, or when the instruction is actually sent to the memory system. Any option works as long as it is used in a consistent fashion.

Example Let's redo the preceding example, but this time we assume the processor with the longer clock cycle time supports out-of-order execution yet still has a direct-mapped cache. Assume 30% of the 65 ns miss penalty can be overlapped; that is, the average CPU memory stall time is now 45.5 ns.

Answer Average memory access time for the out-of-order (OOO) computer is

$$\text{Average memory access time}_{\text{1-way,OOO}} = 0.35 \times 1.35 + (0.021 \times 45.5) = 1.43\,\text{ns}$$

The performance of the OOO cache is

$$\text{CUP time}_{\text{1-way,OOO}} = \text{IC} \times [1.6 \times 0.35 \times 1.35 + (0.021 \times 1.4 \times 45.5)] = 2.09 \times \text{IC}$$

Hence, despite a much slower clock cycle time and the higher miss rate of a direct-mapped cache, the out-of-order computer can be slightly faster if it can hide 30% of the miss penalty.

In summary, although the state of the art in defining and measuring memory stalls for out-of-order processors is complex, be aware of the issues because they significantly affect performance. The complexity arises because out-of-order processors tolerate some latency due to cache misses without hurting performance. Consequently, designers usually use simulators of the out-of-order processor and memory when evaluating trade-offs in the memory hierarchy to be sure that an improvement that helps the average memory latency actually helps program performance.

To help summarize this section and to act as a handy reference, Figure B.7 lists the cache equations in this appendix.

$$
\begin{aligned}
2^{\text{index}} &= \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}\\
\text{CPU execution time} &= (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}\\
\text{Memory stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\
\text{Memory stall cycles} &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\
\frac{\text{Misses}}{\text{Instruction}} &= \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}\\
\text{Average memory access time} &= \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}\\
\text{CPU execution time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}}\right) \times \text{Clock cycle time}\\
\text{CPU execution time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}\\
\text{CPU execution time} &= \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty}\right) \times \text{Clock cycle time}\\
\frac{\text{Memory stall cycles}}{\text{Instruction}} &= \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})\\
\text{Average memory access time} &= \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})\\
\frac{\text{Memory stall cycles}}{\text{Instruction}} &= \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}
\end{aligned}
$$

Figure B.7 Summary of performance equations in this appendix. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

B.3 Six Basic Cache Optimizations

The average memory access time formula gave us a framework to present cache optimizations for improving cache performance:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Hence, we organize six cache optimizations into three categories:

- *Reducing the miss rate*—larger block size, larger cache size, and higher associativity
- *Reducing the miss penalty*—multilevel caches and giving reads priority over writes
- *Reducing the time to hit in the cache*—avoiding address translation when indexing the cache

Figure B.18 on page B-40 concludes this section with a summary of the implementation complexity and the performance benefits of these six techniques.

The classical approach to improving cache behavior is to reduce miss rates, and we present three techniques to do so. To gain better insights into the causes of misses, we first start with a model that sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. These are also called *cold-start misses* or *first-reference misses.*
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses*. The idea is that hits in a fully associative cache that become misses in an *n*-way set-associative cache are due to more than *n* requests on some popular sets.

(Chapter 5 adds a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we won't consider those here.)

Figure B.8 shows the relative frequency of cache misses, broken down by the three C's. Compulsory misses are those that occur in an infinite cache. Capacity misses are those that occur in a fully associative cache. Conflict misses are those that occur going from fully associative to eight-way associative, four-way associative, and so on. Figure B.9 presents the same data graphically. The top graph shows absolute miss rates; the bottom graph plots the percentage of all the misses by type of miss as a function of cache size.

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions of conflict misses and how they are calculated:

- *Eight-way*—Conflict misses due to going from fully associative (no conflicts) to eight-way associative
- *Four-way*—Conflict misses due to going from eight-way associative to four-way associative
- *Two-way*—Conflict misses due to going from four-way associative to two-way associative
- *One-way*—Conflict misses due to going from two-way associative to one-way associative (direct mapped)

As we can see from the figures, the compulsory miss rate of the SPEC2000 programs is very small, as it is for many long-running programs.

Having identified the three C's, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all

Cache size (KiB)	Degree associative	Total miss rate	Miss rate components (relative percent) *(sum = 100% of total miss rate)*					
			Compulsory		Capacity		Conflict	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size *N* has about the same miss rate as a two-way set-associative cache of size *N*/2 up through 128 K. Caches larger than 128 KiB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

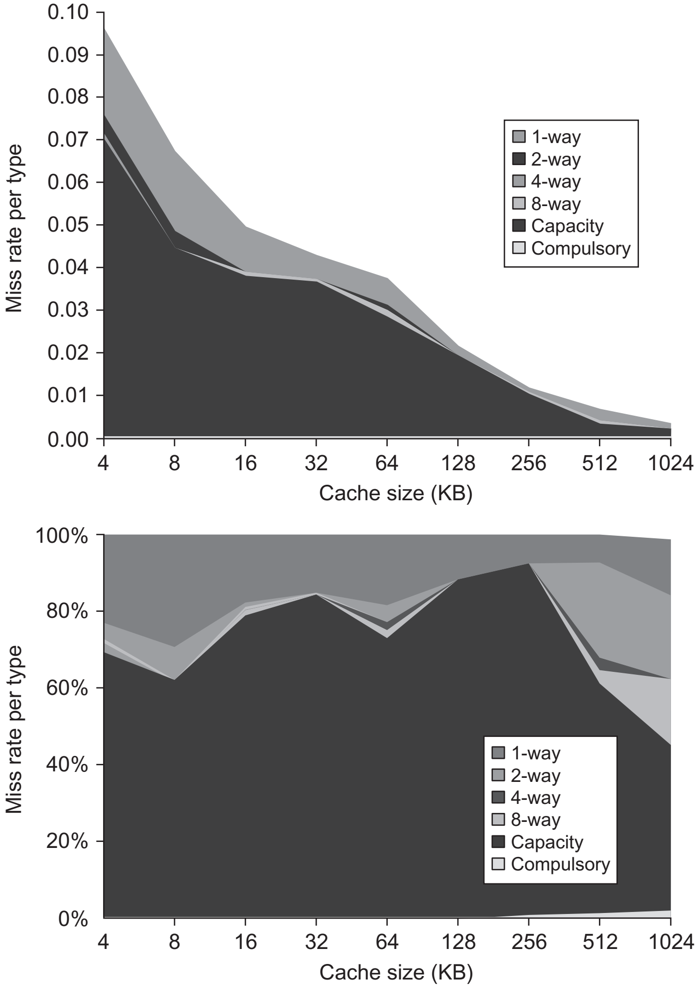


Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (*Space allows* the graphs to show one extra cache size than can fit in Figure B.8.)

conflict misses. Full associativity is expensive in hardware, however, and may slow the processor clock rate (see the example on page B-29), leading to lower overall performance.

There is little to be done about capacity except to enlarge the cache. If the upper-level memory is much smaller than what is needed for a program, and a significant percentage of the time is spent moving data between two levels in the hierarchy, the memory hierarchy is said to *thrash*. Because so many replacements are required, thrashing means the computer runs close to the speed of the lower-level memory, or maybe even slower because of the miss overhead.

Another approach to improving the three C's is to make blocks larger to reduce the number of compulsory misses, but, as we will see shortly, large blocks can increase other kinds of misses.

The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behavior but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, because a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Similarly, changing the block size can sometimes reduce capacity misses (in addition to the expected reduction in compusolory misses), as Gupta et al. (2013) show.

Note also that the three C's also ignore replacement policy, because it is difficult to model and because, in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which contradicts the three C's model. (Some have proposed using an address trace to determine optimal placement in memory to avoid placement misses from the three C's model; we've not followed that advice here.)

Alas, many of the techniques that reduce miss rates also increase hit time or miss penalty. The desirability of reducing miss rates using the three optimizations must be balanced against the goal of making the whole system fast. This first example shows the importance of a balanced perspective.

First Optimization: Larger Block Size to Reduce Miss Rate

The simplest way to reduce miss rate is to increase the block size. Figure B.10 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

At the same time, larger blocks increase the miss penalty. Because they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it *increases* the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure B.11?

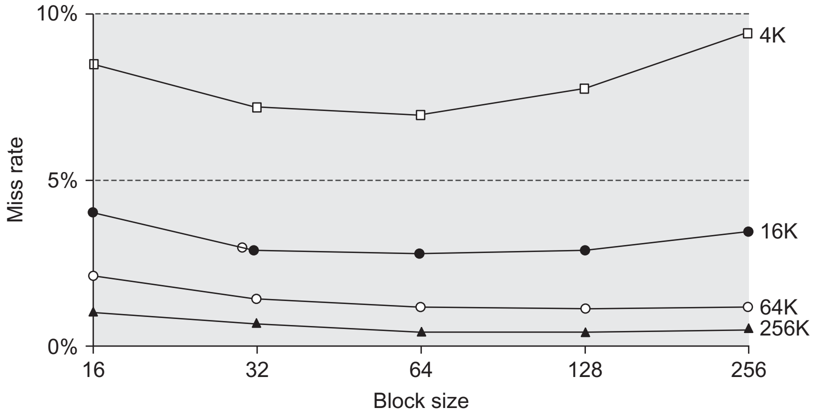


Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al. 1993).

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KiB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

and for a 256-byte block in a 256 KiB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

	Cache size			
Block size	**4K**	**16K**	**64K**	**256K**
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KiB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KiB in order for a 256-byte block to decrease misses.

		Cache size			
Block size	**Miss penalty**	**4K**	**16K**	**64K**	**256K**
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

Figure B.12 shows the average memory access time for all block and cache sizes between those two extremes. The boldfaced entries show the fastest block size for a given cache size: 32 bytes for 4 KiB and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

As in all of these techniques, the cache designer is trying to minimize both the miss rate and the miss penalty. The selection of block size depends on both the latency and bandwidth of the lower-level memory. High latency and high bandwidth encourage large block size because the cache gets many more bytes per miss for a small increase in miss penalty. Conversely, low latency and low bandwidth encourage smaller block sizes because there is little time saved from a larger block. For example, twice the miss penalty of a small block may be close to the penalty of a block twice the size. The larger number of small blocks may also reduce conflict misses. Note that Figures B.10 and B.12 show the difference between selecting a block size based on minimizing miss rate versus minimizing average memory access time.

After seeing the positive and negative impact of larger block size on compulsory and capacity misses, the next two subsections look at the potential of higher capacity and higher associativity.

Second Optimization: Larger Caches to Reduce Miss Rate

The obvious way to reduce capacity misses in Figures B.8 and B.9 is to increase capacity of the cache. The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

Figures B.8 and B.9 show how miss rates improve with higher associativity. There are two general rules of thumb that can be gleaned from these figures. The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the eight-way entries to the capacity miss column in Figure B.8, because capacity misses are calculated using fully associative caches.

The second observation, called the *2:1 cache rule of thumb*, is that a direct-mapped cache of size *N* has about the same miss rate as a two-way set associative cache of size *N*/2. This held in three C's figures for cache sizes less than 128 KiB.

Like many of these examples, improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time. Hence, the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity, as the following example suggests.

Example Assume that higher associativity would increase the clock cycle time as listed as follows:

$$\begin{aligned}\text{Clock cycle time}_{\text{2-way}} &= 1.36 \times \text{Clock cycle time}_{\text{1-way}}\\ \text{Clock cycle time}_{\text{4-way}} &= 1.44 \times \text{Clock cycle time}_{\text{1-way}}\\ \text{Clock cycle time}_{\text{8-way}} &= 1.52 \times \text{Clock cycle time}_{\text{1-way}}\end{aligned}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\begin{aligned}\text{Average memory access time}_{\text{8-way}} &< \text{Average memory access time}_{\text{4-way}}\\ \text{Average memory access time}_{\text{4-way}} &< \text{Average memory access time}_{\text{2-way}}\\ \text{Average memory access time}_{\text{2-way}} &< \text{Average memory access time}_{\text{1-way}}\end{aligned}$$

Answer Average memory access time for each associativity is

$$\begin{aligned}\text{Average memory access time}_{\text{8-way}} &= \text{Hit time}_{\text{8-way}} + \text{Miss rate}_{\text{8-way}} \times \text{Miss penalty}_{\text{8-way}}\\ &= 1.52 + \text{Miss rate}_{\text{8-way}} \times 25\\ \text{Average memory access time}_{\text{4-way}} &= 1.44 + \text{Miss rate}_{\text{4-way}} \times 25\\ \text{Average memory access time}_{\text{2-way}} &= 1.36 + \text{Miss rate}_{\text{2-way}} \times 25\\ \text{Average memory access time}_{\text{1-way}} &= 1.00 + \text{Miss rate}_{\text{1-way}} \times 25\end{aligned}$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4 KiB direct-mapped cache is

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.098 \times 25) = 3.44$$

and the time for a 512 KiB, eight-way set associative cache is

$$\text{Average memory access time}_{\text{8-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure B.8, Figure B.13 shows the average memory access time for each cache and associativity. The figure shows

	Associativity			
Cache size (KiB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. *Boldface* type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

that the formulas in this example hold for caches less than or equal to 8 KiB for up to four-way associativity. Starting with 16 KiB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

Fourth Optimization: Multilevel Caches to Reduce Miss Penalty

Reducing cache misses had been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate. Moreover, Figure 2.2 on page 80 shows that technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.

This performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of processors, or make the cache larger to overcome the widening gap between the processor and main memory?

One answer is, do both. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast processor. Yet, the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Although the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are not always straightforward. Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the original formula is

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times \text{Miss penalty}_{\text{L1}}$$

and

$$\text{Miss penalty}_{\text{L1}} = \text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}$$

so

$$\begin{aligned}\text{Average memory access time} = {} & \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \\ & \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})\end{aligned}$$

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1}, and for the second-level cache it is Miss rate_{L2}.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1}, but for the second-level cache it is Miss $\text{rate}_{\text{L1}} \times$ Miss rate_{L2}.

This local miss rate is large for second-level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: It indicates what fraction of the memory accesses that leave the processor go all the way to memory.

Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second-level cache.

$$\begin{aligned}\text{Average memory stalls per instruction} = {} & \text{Misses per instruction}_{\text{L1}} \times \text{Hit time}_{\text{L2}} \\ & + \text{Misses per instruction}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}\end{aligned}$$

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\begin{aligned}\text{Average memory access time} &= \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}\end{aligned}$$

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40 × 1.5 or 60 L1 misses, and 20 × 1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \\ &\quad \times \text{Miss penalty}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6\ \text{clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6\ \text{clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

Note that these formulas are for combined reads and writes, assuming a write-back first-level cache. Obviously, a write-through first-level cache will send *all* writes to the second level, not just the misses, and a write buffer might be used.

Figures B.14 and B.15 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence, our intuition and knowledge about the first-level caches apply. The second insight is that the local cache miss rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the processor, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There are two major questions for the design of the second-level cache: Will it lower the average memory access time portion of the CPI, and how much does it cost?

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires the design of huge second-level caches—the size of main memory in older computers!

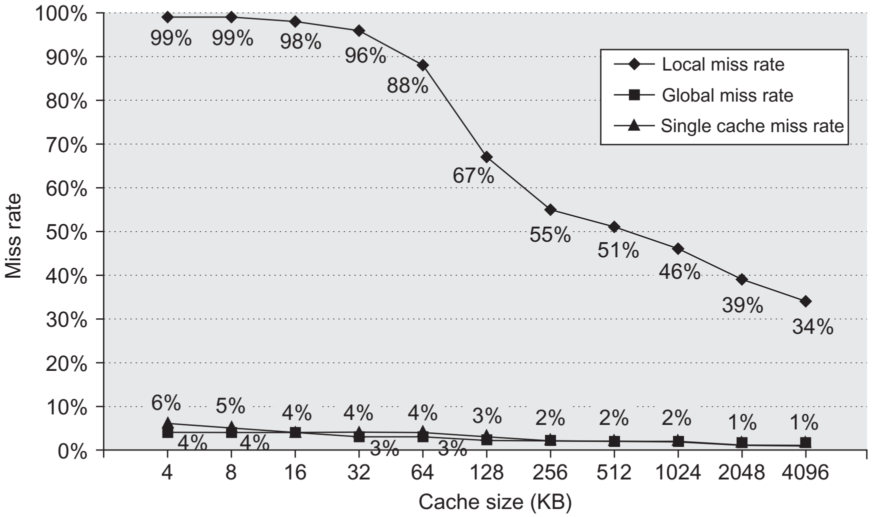


Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KiB first-level caches make little sense, as reflected in the high miss rates. After 256 KiB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KiB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KiB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

One question is whether set associativity makes more sense for second-level caches.

Example Given the following data, what is the impact of second-level cache associativity on its miss penalty?

- Hit time_{L2} for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate_{L2} for direct mapped = 25%.
- Local miss rate_{L2} for two-way set associative = 20%.
- Miss penalty_{L2} = 200 clock cycles.

Answer For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0\,\text{clock cycles}$$

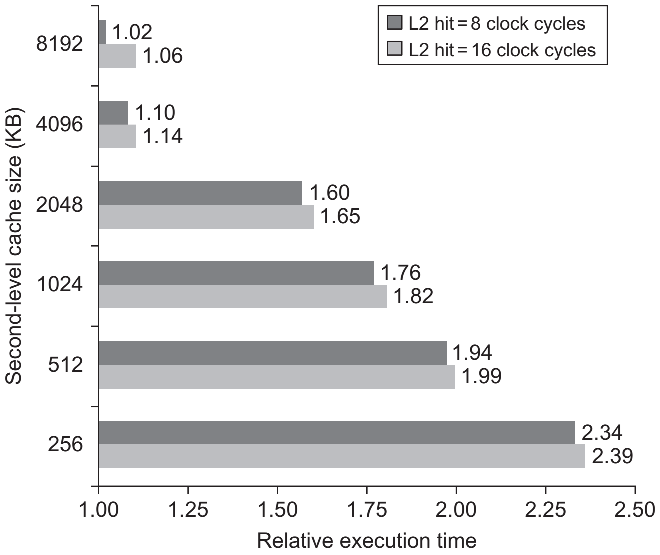


Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KiB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

Now we can reduce the miss penalty by reducing the *miss rate* of the second-level caches.

Another consideration concerns whether data in the first-level cache are in the second-level cache. *Multilevel inclusion* is the natural policy for memory hierarchies: L1 data are always present in L2. Inclusion is desirable because consistency between I/O and caches (or among caches in a multiprocessor) can be determined just by checking the second-level cache.

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache. Inclusion can still be maintained with more work on a second-level miss. The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion:* L1 data are *never* found in an L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of an L1 block with an L2 block. This policy prevents wasting space in the L2 cache. For example, the AMD Opteron chip obeys the exclusion property using two 64 KiB L1 caches and 1 MiB L2 cache.

As these issues illustrate, although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job given a compatible second-level cache. It is less of a gamble to use a write through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes and it uses multilevel inclusion.

The essence of all cache designs is balancing fast hits and few misses. For second-level caches, there are far fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to much larger caches and techniques to lower the miss rate, such as higher associativity and larger blocks.

Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

This optimization serves reads before writes have been completed. We start with looking at the complexities of a write buffer.

With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses because they might hold the updated value of a location needed on a read miss.

Example Look at this code sequence:

```
sd x3, 512(x0);M[512] ¬ R3 (cache index 0)
ld x1, 1024(x0);x1 ¬ M[1024](cache index 0)
ld x2, 512(x0);x2 ¬ M[512] (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in x2 always be equal to the value in x3?

Answer Using the terminology from Chapter 2, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in x3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register x2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into x2. Without proper precautions, x3x1 would not be equal to x2!

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the processor read, for which the processor is probably waiting, will finish sooner. Similar to the previous situation, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Now that we have five optimizations that reduce cache miss penalties or miss rates, it is time to look at reducing the final component of average memory access time. Hit time is critical because it can affect the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

Sixth Optimization: Avoiding Address Translation During Indexing of the Cache to Reduce Hit Time

Even a small and simple cache must cope with the translation of a virtual address from the processor to a physical address to access memory. As described in Section B.4, processors treat main memory as just another level of the memory hierarchy, and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.

The guideline of making the common case fast suggests that we use virtual addresses for the cache, because hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. As we will shortly see, it is important to distinguish two tasks: indexing the cache and comparing addresses. Thus, the issues are whether a virtual or physical address is used to index the cache and whether a virtual or physical address is used in the tag comparison. Full virtual addressing for both indices and tags eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches?

One reason is protection. Page-level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. One solution is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache.

Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure B.16 shows the impact on miss rates of this flushing. One solution is to increase the width of the cache address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is

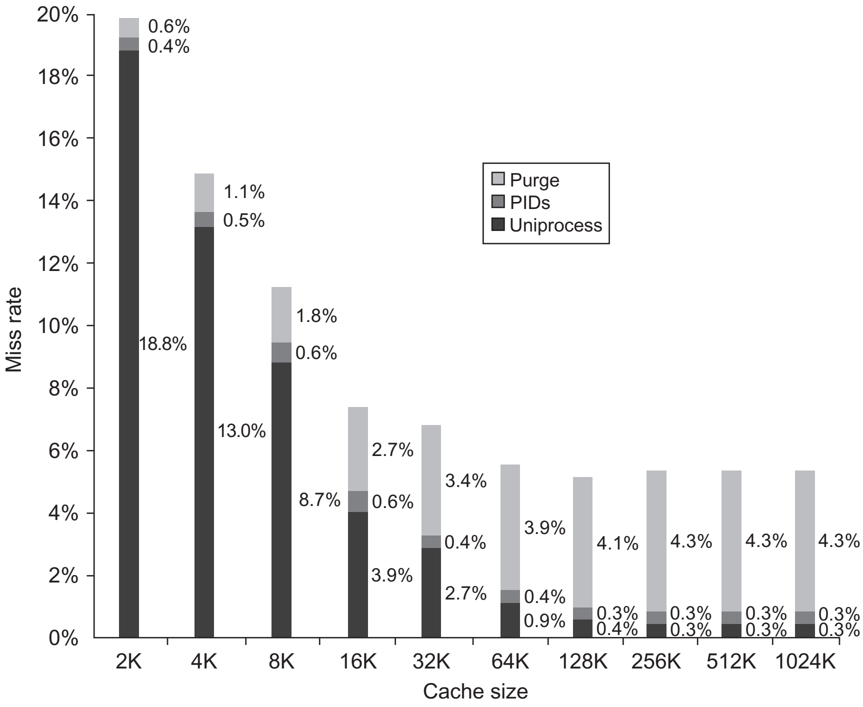


Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3%–0.6% and save 0.6%–4.3% over purging. Agarwal (1987) collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128 to 256 K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

recycled; that is, the PID distinguishes whether or not the data in the cache are for this program. Figure B.16 shows the improvement in miss rates by using PIDs to avoid cache flushes.

A third reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address. These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value. With a physical cache this wouldn't happen, because the accesses would first be translated to the same physical cache block.

Hardware solutions to the synonym problem, called *antialiasing*, guarantee every cache block a unique physical address. For example, the AMD Opteron uses a 64 KiB instruction cache with a 4 KiB page and two-way set associativity; hence, the hardware must handle aliases involved with the three virtual address bits in the set index. It avoids aliases by simply checking all eight possible locations on a miss—two blocks in each of four sets—to be sure that none matches the physical address of the data being fetched. If one is found, it is invalidated, so when the new data are loaded into the cache their physical address is guaranteed to be unique.

Software can make this problem much easier by forcing aliases to share some address bits. An older version of UNIX from Sun Microsystems, for example, required all aliases to be identical in the last 18 bits of their addresses; this restriction is called *page coloring*. Note that page coloring is simply set associative mapping applied to virtual memory: the 4 KiB (2^{12}) pages are mapped using 64 (2^6) sets to ensure that the physical and virtual addresses match in the last 18 bits. This restriction means a direct-mapped cache that is 2^{18} (256 K) bytes or smaller can never have duplicate physical addresses for blocks. From the perspective of the cache, page coloring effectively increases the page offset, as software guarantees that the last few bits of the virtual and physical page address are identical.

The final area of concern with virtual addresses is I/O. I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache. (The impact of I/O on caches is further discussed in Appendix D.)

One alternative to get the best of both virtual and physical caches is to use part of the page offset—the part that is identical in both virtual and physical addresses—to index the cache. At the same time as the cache is being read using that index, the virtual part of the address is translated, and the tag match uses physical addresses.

This alternative allows the cache read to begin immediately, and yet the tag comparison is still with physical addresses. The limitation of this *virtually indexed, physically tagged* alternative is that a direct-mapped cache can be no bigger than the page size. For example, in the data cache in Figure B.5 on page B-13, the index is 9 bits and the cache block offset is 6 bits. To use this trick, the virtual page size would have to be at least $2^{(9+6)}$ bytes or 32 KiB. If not, a portion of the index must be translated from virtual to physical address. Figure B.17 shows the organization of the caches, translation lookaside buffers (TLBs), and virtual memory when this technique is used.

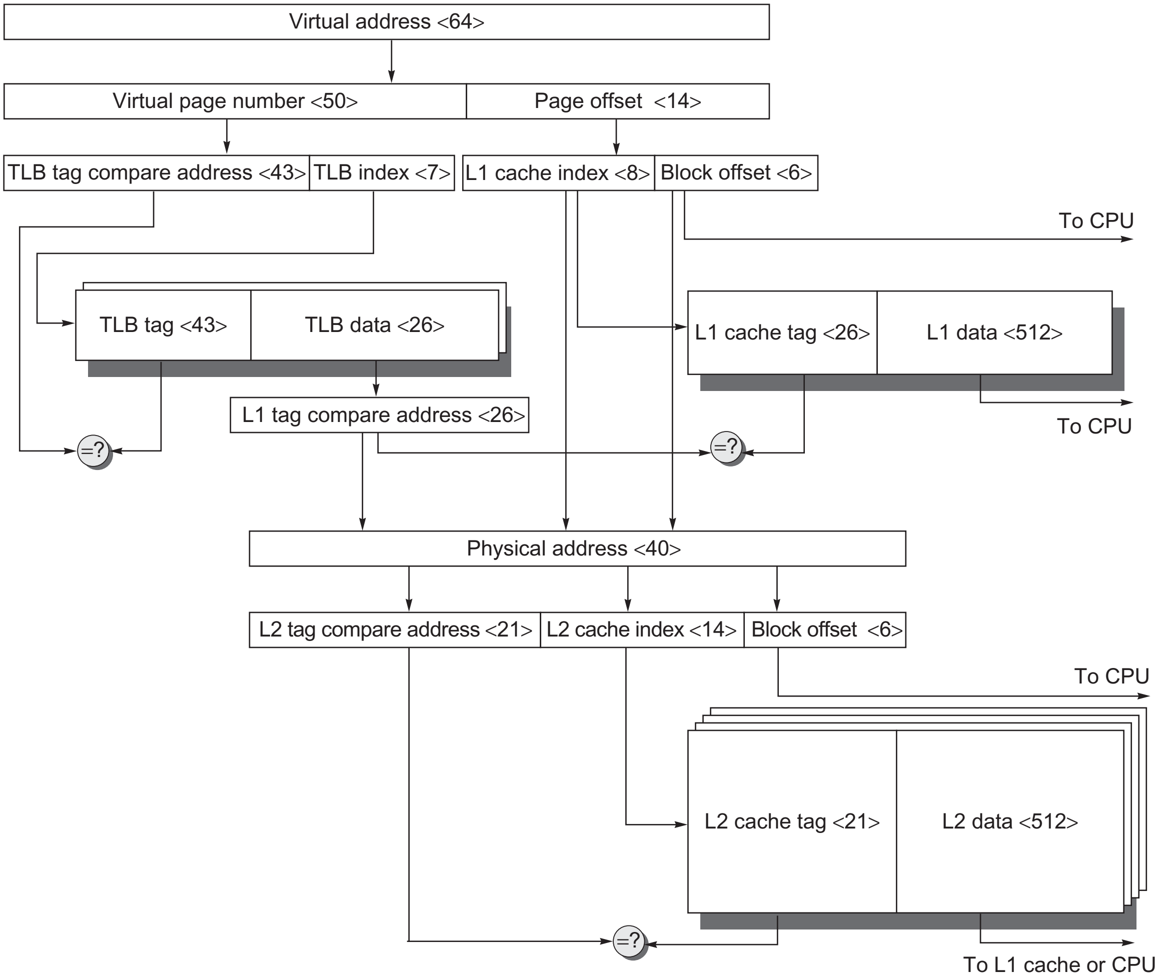

Virtual address <64>
Virtual page number <50>
Page offset <14>
TLB tag compare address <43>
TLB index <7>
L1 cache index <8>
Block offset <6>
To CPU
TLB tag <43>
TLB data <26>
L1 cache tag <26>
L1 data <512>
L1 tag compare address <26>
To CPU
=?
=?
Physical address <40>
L2 tag compare address <21>
L2 cache index <14>
Block offset <6>
To CPU
L2 cache tag <21>
L2 data <512>
=?
To L1 cache or CPU

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KiB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KiB, and the L2 cache is a four-way set associative with a total of 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Associativity can keep the index in the physical part of the address and yet still support a large cache. Recall that the size of the index is controlled by this formula:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

For example, doubling associativity and doubling the cache size does not change the size of the index. The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss rates above

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

8-way set associativity. This high associativity allows a 64 KiB cache to be addressed with a physical index, despite the handicap of 4 KiB pages in the IBM architecture.

Summary of Basic Cache Optimization

The techniques in this section to improve miss rate, miss penalty, and hit time generally impact the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure B.18 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. No optimization in this figure helps more than one category.

B.4 Virtual Memory

> … a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.
>
> Kilburn et al. (1962)

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full address space worth of memory for each process, especially because many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory among many processes.

One way to do this, *virtual memory*, divides physical memory into blocks and allocates them to different processes. Inherent in such an approach must be a *protection* scheme that restricts a process to the blocks belonging only to that process. Most forms of virtual memory also reduce the time to start a program, because not all code and data need be in physical memory before a program can begin.

Although protection provided by virtual memory is essential for current computers, sharing is not the reason that virtual memory was invented. If a program became too large for physical memory, it was the programmer's job to make it fit. Programmers divided programs into pieces, then identified the pieces that were mutually exclusive, and loaded or unloaded these *overlays* under user program control during execution. The programmer ensured that the program never tried to access more physical main memory than was in the computer, and that the proper overlay was loaded at the proper time. As you can well imagine, this responsibility eroded programmer productivity.

Virtual memory was invented to relieve programmers of this burden; it automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage. Figure B.19 shows the mapping of virtual memory to physical memory for a program with four pages.

In addition to sharing protected memory space and automatically managing the memory hierarchy, virtual memory also simplifies loading the program for execution. Called *relocation*, this mechanism allows the same program to run in any location in physical memory. The program in Figure B.19 can be placed anywhere

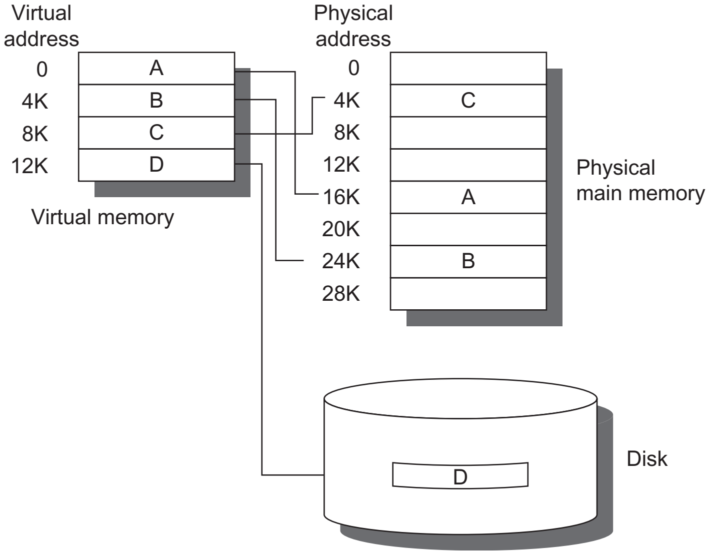


Figure B.19 The logical program in its contiguous virtual address space is shown on the left. It consists of four pages, A, B, C, and D. The actual location of three of the blocks is in physical main memory and the other is located on the disk.

in physical memory or disk just by changing the mapping between them. (Prior to the popularity of virtual memory, processors would include a relocation register just for that purpose.) An alternative to a hardware solution would be software that changed all addresses in a program each time it was run.

Several general memory hierarchy ideas from Chapter 1 about caches are analogous to virtual memory, although many of the terms are different. *Page* or *segment* is used for block, and *page fault* or *address fault* is used for miss. With virtual memory, the processor produces *virtual addresses* that are translated by a combination of hardware and software to *physical addresses*, which access main memory. This process is called *memory mapping* or *address translation*. Today, the two memory hierarchy levels controlled by virtual memory are DRAMs and magnetic disks. Figure B.20 shows a typical range of memory hierarchy parameters for virtual memory.

There are further differences between caches and virtual memory beyond those quantitative ones mentioned in Figure B.20:

- Replacement on cache misses is primarily controlled by hardware, while virtual memory replacement is primarily controlled by the operating system. The longer miss penalty means it's more important to make a good decision, so the operating system can be involved and take time deciding what to replace.
- The size of the processor address determines the size of virtual memory, but the cache size is independent of the processor address size.
- In addition to acting as the lower-level backing store for main memory in the hierarchy, secondary storage is also used for the file system. In fact, the file system occupies most of secondary storage. It is not usually in the address space.

Parameter	**First-level cache**	**Virtual memory**
Block (page) size	16–128 bytes	4096–65,536 bytes
Hit time	1–3 clock cycles	100–200 clock cycles
Miss penalty	8–200 clock cycles	1,000,000–10,000,000 clock cycles
(access time)	(6–160 clock cycles)	(800,000–8,000,000 clock cycles)
(transfer time)	(2–40 clock cycles)	(200,000–2,000,000 clock cycles)
Miss rate	0.1%–10%	0.00001%–0.001%
Address mapping	25–45-bit physical address to 14–20-bit cache address	32–64-bit virtual address to 25–45-bit physical address

Figure B.20 Typical ranges of parameters for caches and virtual memory. Virtual memory parameters represent increases of 10–1,000,000 times over cache parameters. Usually, first-level caches contain at most 1 MiB of data, whereas physical memory contains 256 MiB to 1 TB.

Virtual memory also encompasses several related techniques. Virtual memory systems can be categorized into two classes: those with fixed-size blocks, called *pages*, and those with variable-size blocks, called *segments*. Pages are typically fixed at 4096–8192 bytes, while segment size varies. The largest segment supported on any processor ranges from 2^{16} bytes up to 2^{32} bytes; the smallest segment is 1 byte. Figure B.21 shows how the two approaches might divide code and data.

The decision to use paged virtual memory versus segmented virtual memory affects the processor. Paged addressing has a single fixed-size address divided into page number and offset within a page, analogous to cache addressing. A single address does not work for segmented addresses; the variable size of segments requires 1 word for a segment number and 1 word for an offset within a segment, for a total of 2 words. An unsegmented address space is simpler for the compiler.

The pros and cons of these two approaches have been well documented in operating systems textbooks; Figure B.22 summarizes the arguments. Because of the

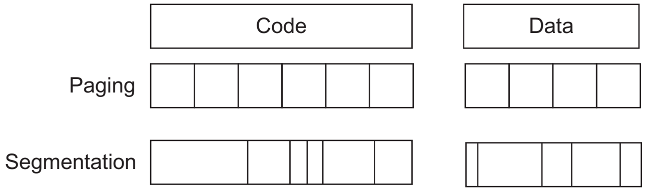


Figure B.21 Example of how paging and segmentation divide a program.

	Page	**Segment**
Words per address	One	Two (segment and offset)
Programmer visible?	Invisible to application programmer	May be visible to application programmer
Replacing a block	Trivial (all blocks are the same size)	Difficult (must find contiguous, variable-size, unused portion of main memory)
Memory use inefficiency	Internal fragmentation (unused portion of page)	External fragmentation (unused pieces of main memory)
Efficient disk traffic	Yes (adjust page size to balance access time and transfer time)	Not always (small segments may transfer just a few bytes)

Figure B.22 Paging versus segmentation. Both can waste memory, depending on the block size and how well the segments fit together in main memory. Programming languages with unrestricted pointers require both the segment and the address to be passed. A hybrid approach, called *paged segments*, shoots for the best of both worlds: segments are composed of pages, so replacing a block is easy, yet a segment may be treated as a logical unit.

replacement problem (the third line of the figure), few computers today use pure segmentation. Some computers use a hybrid approach, called *paged segments*, in which a segment is an integral number of pages. This simplifies replacement because memory need not be contiguous, and the full segments need not be in main memory. A more recent hybrid is for a computer to offer multiple page sizes, with the larger sizes being powers of 2 times the smallest page size. The IBM 405CR embedded processor, for example, allows 1 KiB, 4 KiB ($2^2 \times 1$ KiB), 16 KiB ($2^4 \times 1$ KiB), 64 KiB ($2^6 \times 1$ KiB), 256 KiB ($2^8 \times 1$ KiB), 1024 KiB ($2^{10} \times 1$ KiB), and 4096 KiB ($2^{12} \times 1$ KiB) to act as a single page.

Four Memory Hierarchy Questions Revisited

We are now ready to answer the four memory hierarchy questions for virtual memory.

Q1: Where Can a Block be Placed in Main Memory?

The miss penalty for virtual memory involves access to a rotating magnetic storage device and is therefore quite high. Given the choice of lower miss rates or a simpler placement algorithm, operating systems designers usually pick lower miss rates because of the exorbitant miss penalty. Thus, operating systems allow blocks to be placed anywhere in main memory. According to the terminology in Figure B.2 on page B-8, this strategy would be labeled fully associative.

Q2: How Is a Block Found If It Is in Main Memory?

Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address (see Figure B.23).

This data structure, containing the physical page addresses, usually takes the form of a *page table.* Indexed by the virtual page number, the size of the table is the number of pages in the virtual address space. Given a 32-bit virtual address, 4 KiB pages, and 4 bytes per page table entry (PTE), the size of the page table would be $(2^{32}/2^{12}) \times 2^2 = 2^{22}$ or 4 MiB.

To reduce the size of this data structure, some computers apply a hashing function to the virtual address. The hash allows the data structure to be the length of the number of *physical* pages in main memory. This number could be much smaller than the number of virtual pages. Such a structure is called an *inverted page table*. Using the previous example, a 512 MiB physical memory would only need 1 MiB (8×512 MiB/4 KiB) for an inverted page table; the extra 4 bytes per page table entry are for the virtual address. The HP/Intel IA-64 covers both bases by offering

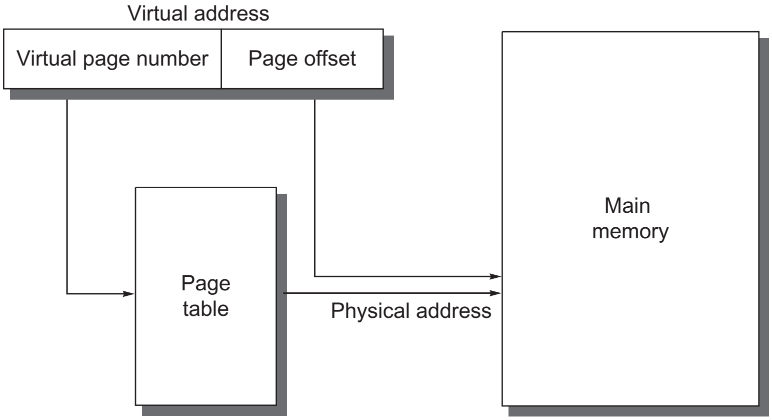


Figure B.23 The mapping of a virtual address to a physical address via a page table.

both traditional pages tables *and* inverted page tables, leaving the choice of mechanism to the operating system programmer.

To reduce address translation time, computers use a cache dedicated to these address translations, called a *translation lookaside buffer*, or simply *translation buffer*, described in more detail shortly.

Q3: Which Block Should be Replaced on a Virtual Memory Miss?

As mentioned earlier, the overriding operating system guideline is minimizing page faults. Consistent with this guideline, almost all operating systems try to replace the least recently used (LRU) block because if the past predicts the future, that is the one less likely to be needed.

To help the operating system estimate LRU, many processors provide a *use bit* or *reference bit*, which is logically set whenever a page is accessed. (To reduce work, it is actually set only on a translation buffer miss, which is described shortly.) The operating system periodically clears the use bits and later records them so it can determine which pages were touched during a particular time period. By keeping track in this way, the operating system can select a page that is among the least recently referenced.

Q4: What Happens on a Write?

The level below main memory contains rotating magnetic disks that take millions of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory operating system that writes through main memory to disk on every store by the processor. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write-back.

Because the cost of an unnecessary access to the next-lower level is so high, virtual memory systems usually include a dirty bit. It allows blocks to be written to disk only if they have been altered since being read from the disk.

Techniques for Fast Address Translation

Page tables are usually so large that they are stored in main memory and are sometimes paged themselves. Paging means that every memory access logically takes at least twice as long, with one memory access to obtain the physical address and a second access to get the data. As mentioned in Chapter 2, we use locality to avoid the extra memory access. By keeping address translations in a special cache, a memory access rarely requires a second access to translate the data. This special address translation cache is referred to as a *translation look aside buffer* (TLB), also called a *translation buffer* (TB).

A TLB entry is like a cache entry where the tag holds portions of the virtual address and the data portion holds a physical page frame number, protection field, valid bit, and usually a use bit and dirty bit. To change the physical page frame number or protection of an entry in the page table, the operating system must make sure the old entry is not in the TLB; otherwise, the system won't behave properly. Note that this dirty bit means the corresponding *page* is dirty, not that the address translation in the TLB is dirty nor that a particular block in the data cache is dirty. The operating system resets these bits by changing the value in the page table and then invalidates the corresponding TLB entry. When the entry is reloaded from the page table, the TLB gets an accurate copy of the bits.

Figure B.24 shows the Opteron data TLB organization, with each step of the translation labeled. This TLB uses fully associative placement; thus, the translation begins (steps 1 and 2) by sending the virtual address to all tags. Of course, the tag must be marked valid to allow a match. At the same time, the type of memory access is checked for a violation (also in step 2) against protection information in the TLB.

For reasons similar to those in the cache case, there is no need to include the 12 bits of the page offset in the TLB. The matching tag sends the corresponding physical address through effectively a 40:1 multiplexor (step 3). The page offset is then combined with the physical page frame to form a full physical address (step 4). The address size is 40 bits.

Address translation can easily be on the critical path determining the clock cycle of the processor, so the Opteron uses virtually addressed, physically tagged L1 caches.

Selecting a Page Size

The most obvious architectural parameter is the page size. Choosing the page is a question of balancing forces that favor a larger page size versus those favoring a smaller size. The following favor a larger size:

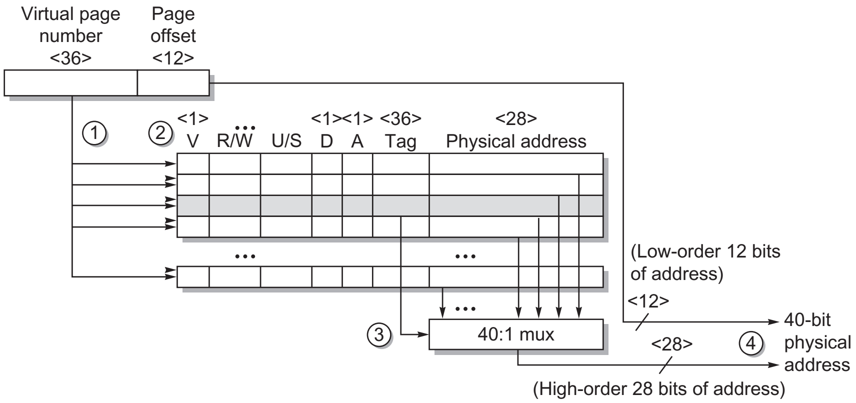


Figure B.24 Operation of the Opteron data TLB during address translation. The four steps of a TLB hit are shown as *circled numbers*. This TLB has 40 entries. Section B.5 describes the various protection and access fields of an Opteron page table entry.

- The size of the page table is inversely proportional to the page size; memory (or other resources used for the memory map) can therefore be saved by making the pages bigger.
- As mentioned in Section B.3, a larger page size can allow larger caches with fast cache hit times.
- Transferring larger pages to or from secondary storage, possibly over a network, is more efficient than transferring smaller pages.
- The number of TLB entries is restricted, so a larger page size means that more memory can be mapped efficiently, thereby reducing the number of TLB misses.

It is for this final reason that recent microprocessors have decided to support multiple page sizes; for some programs, TLB misses can be as significant on CPI as the cache misses.

The main motivation for a smaller page size is conserving storage. A small page size will result in less wasted storage when a contiguous region of virtual memory is not equal in size to a multiple of the page size. The term for this unused memory in a page is *internal fragmentation*. Assuming that each process has three primary segments (text, heap, and stack), the average wasted storage per process will be 1.5 times the page size. This amount is negligible for computers with hundreds of megabytes of memory and page sizes of 4–8 KiB. Of course, when the page sizes become very large (more than 32 KiB), storage (both main and secondary) could be wasted, as well as I/O bandwidth. A final concern is process start-up time; many processes are small, so a large page size would lengthen the time to invoke a process.

Summary of Virtual Memory and Caches

With virtual memory, TLBs, first-level caches, and second-level caches all mapping portions of the virtual and physical address space, it can get confusing what bits go where. Figure B.25 gives a hypothetical example going from a 64-bit virtual address to a 41-bit physical address with two levels of cache. This L1 cache is virtually indexed, and physically tagged because both the cache size and the page size are 8 KiB. The L2 cache is 4 MiB. The block size for both is 64 bytes.

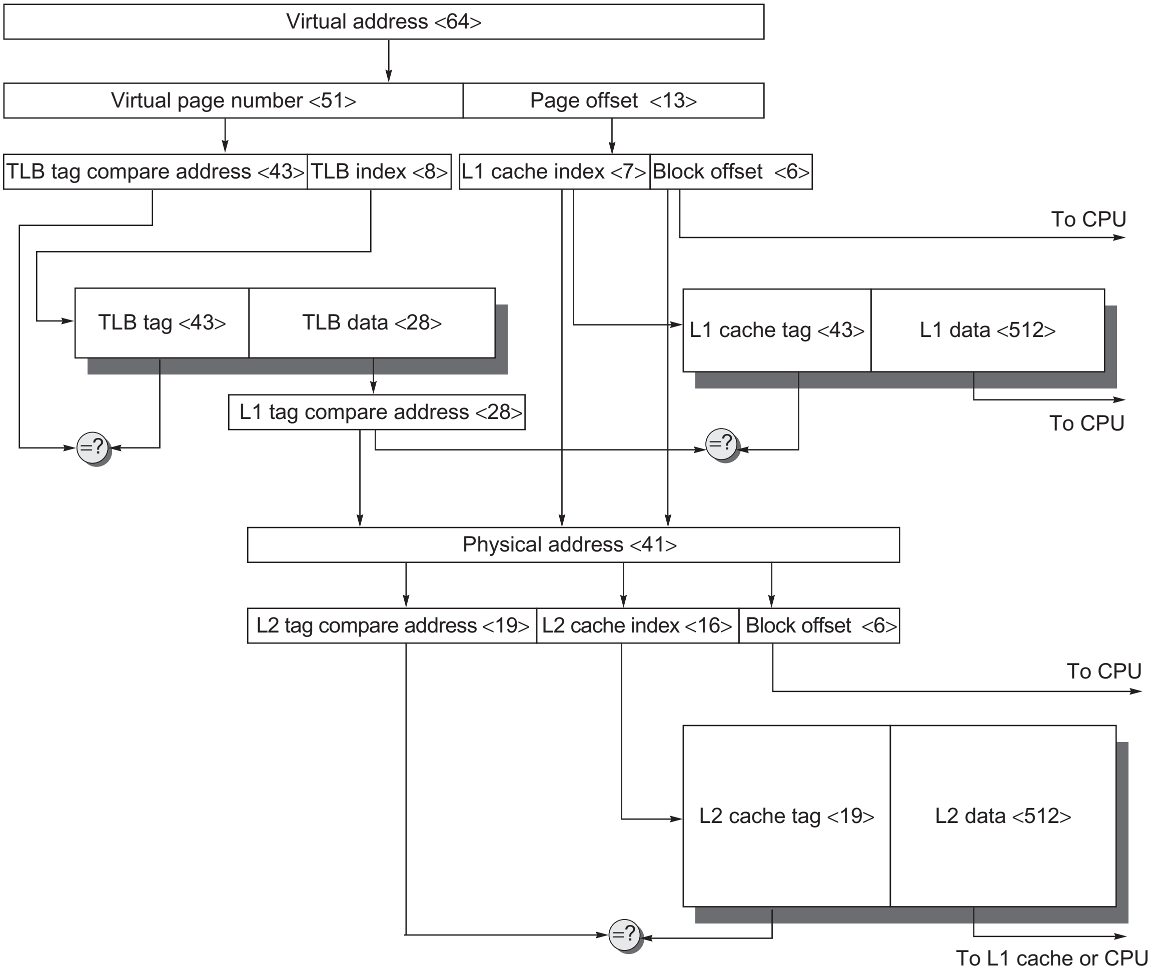

Virtual address <64>
Virtual page number <51>
Page offset <13>
TLB tag compare address <43>
TLB index <8>
L1 cache index <7>
Block offset <6>
To CPU
TLB tag <43>
TLB data <28>
L1 cache tag <43>
L1 data <512>
To CPU
L1 tag compare address <28>
=?
=?
Physical address <41>
L2 tag compare address <19>
L2 cache index <16>
Block offset <6>
To CPU
L2 cache tag <19>
L2 data <512>
=?
To L1 cache or CPU

Figure B.25 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KiB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KiB, and the L2 cache is a direct-mapped 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

First, the 64-bit virtual address is logically divided into a virtual page number and page offset. The former is sent to the TLB to be translated into a physical address, and the high bit of the latter is sent to the L1 cache to act as an index. If the TLB match is a hit, then the physical page number is sent to the L1 cache tag to check for a match. If it matches, it's an L1 cache hit. The block offset then selects the word for the processor.

If the L1 cache check results in a miss, the physical address is then used to try the L2 cache. The middle portion of the physical address is used as an index to the 4 MiB L2 cache. The resulting L2 cache tag is compared with the upper part of the physical address to check for a match. If it matches, we have an L2 cache hit, and the data are sent to the processor, which uses the block offset to select the desired word. On an L2 miss, the physical address is then used to get the block from memory.

Although this is a simple example, the major difference between this drawing and a real cache is replication. First, there is only one L1 cache. When there are two L1 caches, the top half of the diagram is duplicated. Note that this would lead to two TLBs, which is typical. Hence, one cache and TLB is for instructions, driven from the PC, and one cache and TLB is for data, driven from the effective address.

The second simplification is that all the caches and TLBs are direct mapped. If any were n-way set associative, then we would replicate each set of tag memory, comparators, and data memory n times and connect data memories with an n:1 multiplexor to select a hit. Of course, if the total cache size remained the same, the cache index would also shrink by log 2n bits according to the formula in Figure B.7 on page B-22.

B.5 Protection and Examples of Virtual Memory

The invention of multiprogramming, where a computer would be shared by several programs running concurrently, led to new demands for protection and sharing among programs. These demands are closely tied to virtual memory in computers today, and so we cover the topic here along with two examples of virtual memory.

Multiprogramming leads to the concept of a *process*. Metaphorically, a process is a program's breathing air and living space—that is, a running program plus any state needed to continue running it. Time-sharing is a variation of multiprogramming that shares the processor and memory with several interactive users at the same time, giving the illusion that all users have their own computers. Thus, at any instant it must be possible to switch from one process to another. This exchange is called a *process switch* or *context switch*.

A process must operate correctly whether it executes continuously from start to finish, or it is interrupted repeatedly and switched with other processes. The responsibility for maintaining correct process behavior is shared by designers of the computer and the operating system. The computer designer must ensure that the processor portion of the process state can be saved and restored. The operating system designer must guarantee that processes do not interfere with each others' computations.

The safest way to protect the state of one process from another would be to copy the current information to disk. However, a process switch would then take seconds—far too long for a time-sharing environment.

This problem is solved by operating systems partitioning main memory so that several different processes have their state in memory at the same time. This division means that the operating system designer needs help from the computer designer to provide protection so that one process cannot modify another. Besides protection, the computers also provide for sharing of code and data between processes, to allow communication between processes or to save memory by reducing the number of copies of identical information.

Protecting Processes

Processes can be protected from one another by having their own page tables, each pointing to distinct pages of memory. Obviously, user programs must be prevented from modifying their page tables or protection would be circumvented.

Protection can be escalated, depending on the apprehension of the computer designer or the purchaser. *Rings* added to the processor protection structure expand memory access protection from two levels (user and kernel) to many more. Like a military classification system of top secret, secret, confidential, and unclassified, concentric rings of security levels allow the most trusted to access anything, the second most trusted to access everything except the innermost level, and so on. The "civilian" programs are the least trusted and, hence, have the most limited range of accesses. There may also be restrictions on what pieces of memory can contain code—execute protection—and even on the entrance point between the levels. The Intel 80x86 protection structure, which uses rings, is described later in this section. It is not clear whether rings are an improvement in practice over the simple system of user and kernel modes.

As the designer's apprehension escalates to trepidation, these simple rings may not suffice. Restricting the freedom given a program in the inner sanctum requires a new classification system. Instead of a military model, the analogy of this system is to keys and locks: a program can't unlock access to the data unless it has the key. For these keys, or *capabilities*, to be useful, the hardware and operating system must be able to explicitly pass them from one program to another without allowing a program itself to forge them. Such checking requires a great deal of hardware support if time for checking keys is to be kept low.

The 80x86 architecture has tried several of these alternatives over the years. Because backward compatibility is one of the guidelines of this architecture, the most recent versions of the architecture include all of its experiments in virtual memory. We'll go over two of the options here: first the older segmented address space and then the newer flat, 64-bit address space.

A Segmented Virtual Memory Example: Protection in the Intel Pentium

> The second system is the most dangerous system a man ever designs… . The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.
>
> **F. P. Brooks, Jr.**
> *The Mythical Man-Month* (1975)

The original 8086 used segments for addressing, yet it provided nothing for virtual memory or for protection. Segments had base registers but no bound registers and no access checks, and before a segment register could be loaded the corresponding segment had to be in physical memory. Intel's dedication to virtual memory and protection is evident in the successors to the 8086, with a few fields extended to support larger addresses. This protection scheme is elaborate, with many details carefully designed to try to avoid security loopholes. We'll refer to it as IA-32. The next few pages highlight a few of the Intel safeguards; if you find the reading difficult, imagine the difficulty of implementing them!

The first enhancement is to double the traditional two-level protection model: the IA-32 has four levels of protection. The innermost level (0) corresponds to the traditional kernel mode, and the outermost level (3) is the least privileged mode. The IA-32 has separate stacks for each level to avoid security breaches between the levels. There are also data structures analogous to traditional page tables that contain the physical addresses for segments, as well as a list of checks to be made on translated addresses.

The Intel designers did not stop there. The IA-32 divides the address space, allowing both the operating system and the user access to the full space. The IA-32 user can call an operating system routine in this space and even pass parameters to it while retaining full protection. This safe call is not a trivial action, because the stack for the operating system is different from the user's stack. Moreover, the IA-32 allows the operating system to maintain the protection level of the *called* routine for the parameters that are passed to it. This potential loophole in protection is prevented by not allowing the user process to ask the operating system to access something indirectly that it would not have been able to access itself. (Such security loopholes are called *Trojan horses*.)

The Intel designers were guided by the principle of trusting the operating system as little as possible, while supporting sharing and protection. As an example of the use of such protected sharing, suppose a payroll program writes checks and also updates the year-to-date information on total salary and benefits payments. Thus, we want to give the program the ability to read the salary and year-to-date information and modify the year-to-date information but not the salary. We will see the mechanism to support such features shortly. In the rest of this subsection, we will look at the big picture of the IA-32 protection and examine its motivation.

Adding Bounds Checking and Memory Mapping

The first step in enhancing the Intel processor was getting the segmented addressing to check bounds as well as supply a base. Rather than a base address, the segment registers in the IA-32 contain an index to a virtual memory data structure called a *descriptor table*. Descriptor tables play the role of traditional page tables. On the IA-32 the equivalent of a page table entry is a *segment descriptor*. It contains fields found in PTEs:

- *Present bit*—Equivalent to the PTE valid bit, used to indicate this is a valid translation
- *Base field*—Equivalent to a page frame address, containing the physical address of the first byte of the segment
- *Access bit*—Like the reference bit or use bit in some architectures that is helpful for replacement algorithms
- *Attributes field*—Specifies the valid operations and protection levels for operations that use this segment

There is also a *limit field,* not found in paged systems, which establishes the upper bound of valid offsets for this segment. Figure B.26 shows examples of IA-32 segment descriptors.

IA-32 provides an optional paging system in addition to this segmented addressing. The upper portion of the 32-bit address selects the segment descriptor, and the middle portion is an index into the page table selected by the descriptor. The following section describes the protection system that does not rely on paging.

Adding Sharing and Protection

To provide for protected sharing, half of the address space is shared by all processes and half is unique to each process, called *global address space* and *local address space*, respectively. Each half is given a descriptor table with the appropriate name. A descriptor pointing to a shared segment is placed in the global descriptor table, while a descriptor for a private segment is placed in the local descriptor table.

A program loads an IA-32 segment register with an index to the table *and* a bit saying which table it desires. The operation is checked according to the attributes in the descriptor, the physical address being formed by adding the offset in the processor to the base in the descriptor, provided the offset is less than the limit field. Every segment descriptor has a separate 2-bit field to give the legal access level of this segment. A violation occurs only if the program tries to use a segment with a lower protection level in the segment descriptor.

We can now show how to invoke the payroll program mentioned herein to update the year-to-date information without allowing it to update salaries. The program could be given a descriptor to the information that has the writable field clear, meaning it can read but not write the data. A trusted program can then be supplied that will only write the year-to-date information. It is given a descriptor with the

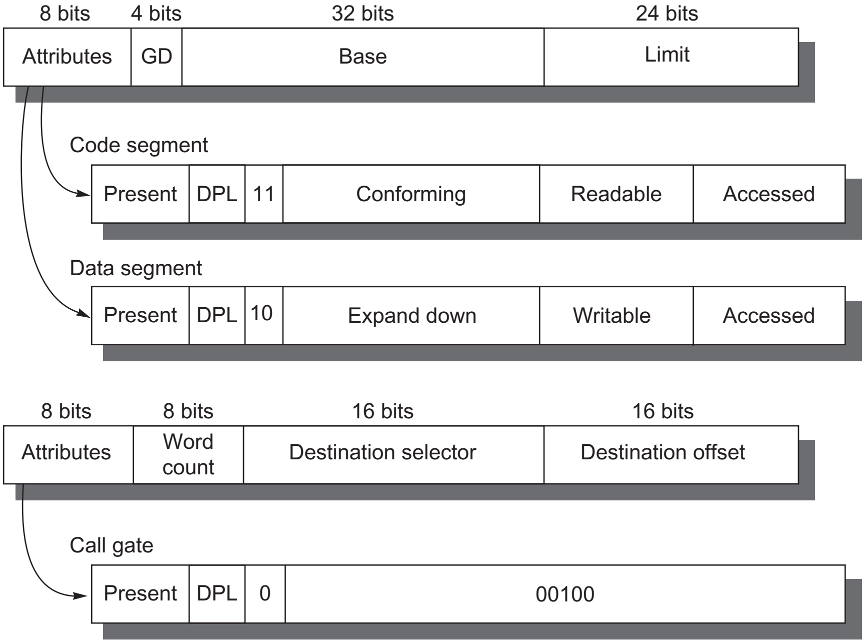

8 bits
4 bits
32 bits
24 bits
Attributes
GD
Base
Limit
Code segment
Present
DPL
11
Conforming
Readable
Accessed
Data segment
Present
DPL
10
Expand down
Writable
Accessed
8 bits
8 bits
16 bits
16 bits
Attributes
Word count
Destination selector
Destination offset
Call gate
Present
DPL
0
00100

Figure B.26 The IA-32 segment descriptors are distinguished by bits in the attributes field. *Base, limit, present, readable*, and *writable* are all self-explanatory. D gives the default addressing size of the instructions: 16 bits or 32 bits. G gives the granularity of the segment limit: 0 means in bytes and 1 means in 4 KiB pages. G is set to 1 when paging is turned on to set the size of the page tables. DPL means *descriptor privilege level*—this is checked against the code privilege level to see if the access will be allowed. *Conforming* says the code takes on the privilege level of the code being called rather than the privilege level of the caller; it is used for library routines. The *expand-down field* flips the check to let the base field be the high-water mark and the limit field be the low-water mark. As you might expect, this is used for stack segments that grow down. *Word count* controls the number of words copied from the current stack to the new stack on a call gate. The other two fields of the call gate descriptor, *destination selector* and *destination offset,* select the descriptor of the destination of the call and the offset into it, respectively. There are many more than these three segment descriptors in the IA-32 protection model.

writable field set (Figure B.26). The payroll program invokes the trusted code using a code segment descriptor with the conforming field set. This setting means the called program takes on the privilege level of the code being called rather than the privilege level of the caller. Hence, the payroll program can read the salaries and call a trusted program to update the year-to-date totals, yet the payroll program cannot modify the salaries. If a Trojan horse exists in this system, to be effective it must be located in the trusted code whose only job is to update the year-to-date information. The argument for this style of protection is that limiting the scope of the vulnerability enhances security.

Adding Safe Calls from User to OS Gates and Inheriting Protection Level for Parameters

Allowing the user to jump into the operating system is a bold step. How, then, can a hardware designer increase the chances of a safe system without trusting the operating system or any other piece of code? The IA-32 approach is to restrict where the user can enter a piece of code, to safely place parameters on the proper stack, and to make sure the user parameters don't get the protection level of the called code.

To restrict entry into others' code, the IA-32 provides a special segment descriptor, or *call gate*, identified by a bit in the attributes field. Unlike other descriptors, call gates are full physical addresses of an object in memory; the offset supplied by the processor is ignored. As stated previously, their purpose is to prevent the user from randomly jumping anywhere into a protected or more privileged code segment. In our programming example, this means the only place the payroll program can invoke the trusted code is at the proper boundary. This restriction is needed to make conforming segments work as intended.

What happens if caller and callee are "mutually suspicious," so that neither trusts the other? The solution is found in the word count field in the bottom descriptor in Figure B.26. When a call instruction invokes a call gate descriptor, the descriptor copies the number of words specified in the descriptor from the local stack onto the stack corresponding to the level of this segment. This copying allows the user to pass parameters by first pushing them onto the local stack. The hardware then safely transfers them onto the correct stack. A return from a call gate will pop the parameters off both stacks and copy any return values to the proper stack. Note that this model is incompatible with the current practice of passing parameters in registers.

This scheme still leaves open the potential loophole of having the operating system use the user's address, passed as parameters, with the operating system's security level, instead of with the user's level. The IA-32 solves this problem by dedicating 2 bits in every processor segment register to the *requested protection level*. When an operating system routine is invoked, it can execute an instruction that sets this 2-bit field in all address parameters with the protection level of the user that called the routine. Thus, when these address parameters are loaded into the segment registers, they will set the requested protection level to the proper value. The IA-32 hardware then uses the requested protection level to prevent any foolishness: no segment can be accessed from the system routine using those parameters if it has a more privileged protection level than requested.

A Paged Virtual Memory Example: The 64-Bit Opteron Memory Management

AMD engineers found few uses of the elaborate protection model described in the previous section. The popular model is a flat, 32-bit address space, introduced by the 80386, which sets all the base values of the segment registers to zero. Hence, AMD dispensed with the multiple segments in the 64-bit mode. It assumes that the segment base is zero and ignores the limit field. The page sizes are 4 KiB, 2 MiB, and 4 MiB.

The 64-bit virtual address of the AMD64 architecture is mapped onto 52-bit physical addresses, although implementations can implement fewer bits to simplify hardware. The Opteron, for example, uses 48-bit virtual addresses and 40-bit physical addresses. AMD64 requires that the upper 16 bits of the virtual address be just the sign extension of the lower 48 bits, which it calls *canonical form*.

The size of page tables for the 64-bit address space is alarming. Hence, AMD64 uses a multilevel hierarchical page table to map the address space to keep the size reasonable. The number of levels depends on the size of the virtual address space. Figure B.27 shows the four-level translation of the 48-bit virtual addresses of the Opteron.

The offsets for each of these page tables come from four 9-bit fields. Address translation starts with adding the first offset to the page-map level 4 base register and then reading memory from this location to get the base of the next-level page table. The next address offset is in turn added to this newly fetched address, and

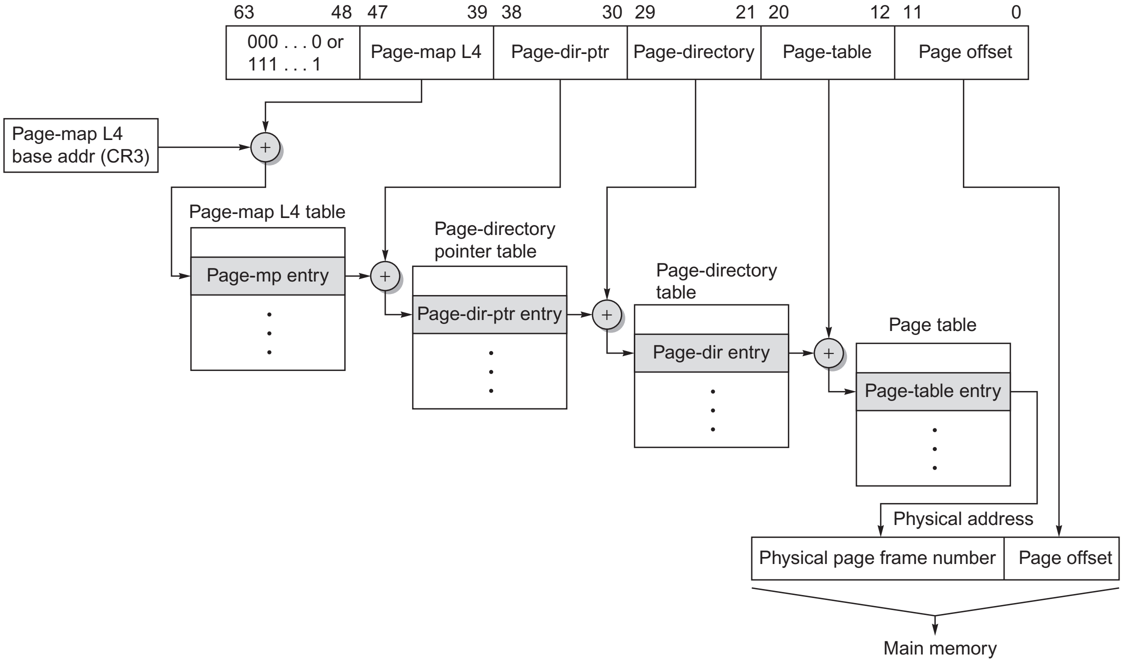

63
48
47
39
38
30
29
21
20
12
11
0
000 . . . 0 or 111 . . . 1
Page-map L4
Page-dir-ptr
Page-directory
Page-table
Page offset
Page-map L4 base addr (CR3)
+
Page-map L4 table
Page-mp entry
Page-directory pointer table
Page-dir-ptr entry
Page-directory table
Page-dir entry
Page table
Page-table entry
Physical address
Physical page frame number
Page offset
Main memory

Figure B.27 The mapping of an Opteron virtual address. The Opteron virtual memory implementation with four page table levels supports an effective physical address size of 40 bits. Each page table has 512 entries, so each level field is 9 bits wide. The AMD64 architecture document allows the virtual address size to grow from the current 48 bits to 64 bits, and the physical address size to grow from the current 40 bits to 52 bits.

memory is accessed again to determine the base of the third page table. It happens again in the same fashion. The last address field is added to this final base address, and memory is read using this sum to (finally) get the physical address of the page being referenced. This address is concatenated with the 12-bit page offset to get the full physical address. Note that the page table in the Opteron architecture fits within a single 4 KiB page.

The Opteron uses a 64-bit entry in each of these page tables. The first 12 bits are reserved for future use, the next 52 bits contain the physical page frame number, and the last 12 bits give the protection and use information. Although the fields vary some between the page table levels, here are the basic ones:

- *Presence*—Says that page is present in memory.
- *Read/write*—Says whether page is read-only or read-write.
- *User/supervisor*—Says whether a user can access the page or if it is limited to the upper three privilege levels.
- *Dirty*—Says if page has been modified.
- *Accessed*—Says if page has been read or written since the bit was last cleared.
- *Page size*—Says whether the last level is for 4 KiB pages or 4 MiB pages; if it's the latter, then the Opteron only uses three instead of four levels of pages.
- *No execute*—Not found in the 80386 protection scheme, this bit was added to prevent code from executing in some pages.
- *Page level cache disable*—Says whether the page can be cached or not.
- *Page level write through*—Says whether the page allows write back or write through for data caches.

Because the Opteron usually goes through four levels of tables on a TLB miss, there are three potential places to check protection restrictions. The Opteron obeys only the bottom-level PTE, checking the others only to be sure the valid bit is set.

As the entry is 8 bytes long, each page table has 512 entries, and the Opteron has 4 KiB pages, the page tables are exactly one page long. Each of the four level fields are 9 bits long, and the page offset is 12 bits. This derivation leaves $64-(4\times 9+12)$ or 16 bits to be sign extended to ensure canonical addresses.

Although we have explained translation of legal addresses, what prevents the user from creating illegal address translations and getting into mischief? The page tables themselves are protected from being written by user programs. Thus, the user can try any virtual address, but by controlling the page table entries the operating system controls what physical memory is accessed. Sharing of memory between processes is accomplished by having a page table entry in each address space point to the same physical memory page.

The Opteron employs four TLBs to reduce address translation time, two for instruction accesses and two for data accesses. Like multilevel caches, the Opteron

Parameter	Description
Block size	1 PTE (8 bytes)
L1 hit time	1 clock cycle
L2 hit time	7 clock cycles
L1 TLB size	Same for instruction and data TLBs: 40 PTEs per TLBs, with 32 4 KiB pages and 8 for 2 MiB or 4 MiB pages
L2 TLB size	Same for instruction and data TLBs: 512 PTEs of 4 KiB pages
Block selection	LRU
Write strategy	(Not applicable)
L1 block placement	Fully associative
L2 block placement	4-way set associative

Figure B.28 Memory hierarchy parameters of the Opteron L1 and L2 instruction and data TLBs.

reduces TLB misses by having two larger L2 TLBs: one for instructions and one for data. Figure B.28 describes the data TLB.

Summary: Protection on the 32-Bit Intel Pentium Versus the 64-Bit AMD Opteron

Memory management in the Opteron is typical of most desktop or server computers today, relying on page-level address translation and correct operation of the operating system to provide safety to multiple processes sharing the computer. Although presented as alternatives, Intel has followed AMD's lead and embraced the AMD64 architecture. Hence, both AMD and Intel support the 64-bit extension of 80x86; yet, for compatibility reasons, both support the elaborate segmented protection scheme.

If the segmented protection model looks harder to build than the AMD64 model, that's because it is. This effort must be especially frustrating for the engineers, because few customers use the elaborate protection mechanism. In addition, the fact that the protection model is a mismatch to the simple paging protection of UNIX-like systems means it will be used only by someone writing an operating system especially for this computer, which hasn't happened yet.

B.6 Fallacies and Pitfalls

Even a review of memory hierarchy has fallacies and pitfalls!

Pitfall *Too small an address space.*

Just five years after DEC and Carnegie Mellon University collaborated to design the new PDP-11 computer family, it was apparent that their creation had a fatal flaw. An architecture announced by IBM six years *before* the PDP-11 was still thriving, with minor modifications, 25 years later. And the DEC VAX, criticized for including unnecessary functions, sold millions of units after the PDP-11 went out of production. Why?

The fatal flaw of the PDP-11 was the size of its addresses (16 bits) as compared with the address sizes of the IBM 360 (24–31 bits) and the VAX (32 bits). Address size limits the program length, because the size of a program and the amount of data needed by the program must be less than $2^{\text{Address size}}$. The reason the address size is so hard to change is that it determines the minimum width of anything that can contain an address: PC, register, memory word, and effective-address arithmetic. If there is no plan to expand the address from the start, then the chances of successfully changing address size are so slim that it usually means the end of that computer family. Bell and Strecker (1976) put it like this:

> There is only one mistake that can be made in computer design that is difficult to recover from—not having enough address bits for memory addressing and memory management. The PDP-11 followed the unbroken tradition of nearly every known computer. [p. 2]

A partial list of successful computers that eventually starved to death for lack of address bits includes the PDP-8, PDP-10, PDP-11, Intel 8080, Intel 8086, Intel 80186, Intel 80286, Motorola 6800, AMI 6502, Zilog Z80, CRAY-1, and CRAY X-MP.

The venerable 80x86 line bears the distinction of having been extended twice, first to 32 bits with the Intel 80386 in 1985 and recently to 64 bits with the AMD Opteron.

Pitfall *Ignoring the impact of the operating system on the performance of the memory hierarchy.*

Figure B.29 shows the memory stall time due to the operating system spent on three large workloads. About 25% of the stall time is either spent in misses in the operating system or results from misses in the application programs because of interference with the operating system.

Pitfall *Relying on the operating systems to change the page size over time.*

The Alpha architects had an elaborate plan to grow the architecture over time by growing its page size, even building it into the size of its virtual address. When it came time to grow page sizes with later Alphas, the operating system designers balked and the virtual memory system was revised to grow the address space while maintaining the 8 KiB page.

Architects of other computers noticed very high TLB miss rates, and so added multiple, larger page sizes to the TLB. The hope was that operating systems programmers would allocate an object to the largest page that made sense, thereby preserving TLB entries. After a decade of trying, most operating systems use these “superpages” only for handpicked functions: mapping the display memory or other I/O devices, or using very large pages for the database code.

			Time						
	Misses		% Time due to application misses		% Time due directly to OS misses				
Workload	% in applications	% in OS	Inherent application misses	OS conflicts with applications	OS instruction misses	Data misses for migration	Data misses in block operations	Rest of OS misses	% Time OS misses and application conflicts
Pmake	47%	53%	14.1%	4.8%	10.9%	1.0%	6.2%	2.9%	25.8%
Multipgm	53%	47%	21.6%	3.4%	9.2%	4.2%	4.7%	3.4%	24.9%
Oracle	73%	27%	25.7%	10.2%	10.6%	2.6%	0.6%	2.8%	26.8%

Figure B.29 Misses and time spent in misses for applications and operating system. The operating system adds about 25% to the execution time of the application. Each processor has a 64 KiB instruction cache and a two-level data cache with 64 KiB in the first level and 256 KiB in the second level; all caches are direct mapped with 16-byte blocks. Collected on Silicon Graphics POWER station 4D/340, a multiprocessor with four 33 MHz R3000 processors running three application workloads under a UNIX System V—Pmake, a parallel compile of 56 files; Multipgm, the parallel numeric program MP3D running concurrently with Pmake and a five-screen edit session; and Oracle, running a restricted version of the TP-1 benchmark using the Oracle database. Data from Torrellas, J., Gupta, A., Hennessy, J., 1992. Characterizing the caching and synchronization performance of a multiprocessor operating system. In: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston (SIGPLAN Notices 27:9 (September)), pp. 162–174.

B.7 Concluding Remarks

The difficulty of building a memory system to keep pace with faster processors is underscored by the fact that the raw material for main memory is the same as that found in the cheapest computer. It is the principle of locality that helps us here—its soundness is demonstrated at all levels of the memory hierarchy in current computers, from disks to TLBs.

However, the increasing relative latency to memory, taking hundreds of clock cycles in 2016, means that programmers and compiler writers must be aware of the parameters of the caches and TLBs if they want their programs to perform well.

B.8 Historical Perspective and References

In Section M.3 (available online) we examine the history of caches, virtual memory, and virtual machines. (The historical section covers both this appendix and Chapter 3.) IBM plays a prominent role in the history of all three. References for further reading are included.

Additional reference: Gupta, S. Xiang, P., Yang, Y., Zhou, H., Locality principle revisited: a probability-based quantitative approach. J. Parallel Distrib. Comput. 73 (7), 1011–1027.

Exercises by Amr Zaky

B.1 [10/10/10/15] <B.1> You are trying to appreciate how important the principle of locality is in justifying the use of a cache memory, so you experiment with a computer having an L1 data cache and a main memory (you exclusively focus on data accesses). The latencies (in CPU cycles) of the different kinds of accesses are as follows: cache hit, 1 cycle; cache miss, 110 cycles; main memory access with cache disabled, 105 cycles.

a. [10] <B.1> When you run a program with an overall miss rate of 3%, what will the average memory access time (in CPU cycles) be?

b. [10] <B.1> Next, you run a program specifically designed to produce completely random data addresses with no locality. Toward that end, you use an array of size 1 GB (all of which fits in the main memory). Accesses to random elements of this array are continuously made (using a uniform random number generator to generate the elements indices). If your data cache size is 64 KB, what will the average memory access time be?

c. [10] <B.1> If you compare the result obtained in part (b) with the main memory access time when the cache is disabled, what can you conclude about the role of the principle of locality in justifying the use of cache memory?

d. [15] <B.1> You observed that a cache hit produces a gain of 104 cycles (1 cycle vs. 105), but it produces a loss of 5 cycles in the case of a miss (110 cycles vs. 105). In the general case, we can express these two quantities as G (gain) and L (loss). Using these two quantities (G and L), identify the highest miss rate after which the cache use would be disadvantageous.

B.2 [15/15] <B.1> For the purpose of this exercise, we assume that we have a 512-byte cache with 64-byte blocks. We will also assume that the main memory is 2 KB large. We can regard the memory as an array of 64-byte blocks: M0, M1, …, M31. Figure B.30 sketches the memory blocks that can reside in different cache blocks if the cache was direct-mapped.

a. [15] <B.1> Show the contents of the table if the cache is organized as a fully-associative cache.

b. [15] <B.1> Repeat part (a) with the cache organized as a four-way set associative cache.

B.3 [10/10/10/10/15/10/15/20] <B.1> Cache organization is often influenced by the desire to reduce the cache's power consumption. For that purpose we assume that the cache is physically distributed into a data array (holding the data), a tag array (holding the tags), and replacement array (holding information needed by replacement policy). Furthermore, every one of these arrays is physically distributed into multiple subarrays (one per way) that can be individually accessed; for example, a four-way set associative least recently used (LRU) cache would have four data subarrays, four tag subarrays, and four replacement subarrays. We assume that the

Cache block	Set	Way	Possible memory blocks
0	0	0	M0, M8, M16, M24
1	1	0	M1, M9, M17, M25
2	2	0	M2, M10, M18, M26
3	3	0	….
4	4	0	….
5	5	0	….
6	6	0	….
7	7	0	M7, M15, M23, M31

Figure B.30 Memory blocks distributed to direct-mapped cache.

Array	Power consumption weight (per way accessed)
Data array	20 units
Tag	Array 5 units
Miscellaneous array	1 unit
Memory access	200 units

Figure B.31 Power consumption costs of different operations.

replacement subarrays are accessed once per access when the LRU replacement policy is used, and once per miss if the first-in, first-out (FIFO) replacement policy is used. It is not needed when a random replacement policy is used. For a specific cache, it was determined that the accesses to the different arrays have the following power consumption weights (Figure B.31):

a. [10] <B.1> A cache read hit. All arrays are read simultaneously.

b. [10] <B.1> Repeat part (a) for a cache read miss.

c. [10] <B.1> Repeat part (a) assuming that the cache access is split across two cycles. In the first cycle, all the tag subarrays are accessed. In the second cycle, only the subarray whose tag matched will be accessed.

d. [10] <B.1> Repeat part (c) for a cache read miss (no data array accesses in the second cycle).

e. [15] <B.1> Repeat part (c) assuming that logic is added to predict the cache way to be accessed. Only the tag subarray for the predicted way is accessed in cycle one. A way hit (address match in predicted way) implies a cache hit. A way miss dictates examining all the tag subarrays in the second cycle. In case of a way hit, only one data subarray (the one whose tag matched) is accessed in cycle two. Assume the way predictor hits.

f. [10] <B.1> Repeat part (e) assuming that the way predictor misses (the way it choses is wrong). When it fails, the way predictor adds an extra cycle in which it accesses all the tag subarrays. Assume the way predictor miss is followed by a cache read hit.

g. [15] <B.1> Repeat part (f) assuming a cache read miss.

h. [20] <B.1> Use parts (e), (f), and (g) for the general case where the workload has the following statistics: way predictor miss rate = 5% and cache miss rate = 3%. (Consider different replacement policies.)

Estimate the memory system (cache + memory) power usage (in power units) for the following configurations. We assume the cache is four-way set associative. Provide answers for the LRU, FIFO, and random replacement policies.

B.4 [10/10/15/15/15/20] <B.1> We compare the write bandwidth requirements of write-through versus write-back caches using a concrete example. Let us assume that we have a 64 KB cache with a line size of 32 bytes. The cache will allocate a line on a write miss. If configured as a write-back cache, it will write back all of the dirty line if it needs to be replaced. We will also assume that the cache is connected to the lower level in the hierarchy through a 64-bit-wide (8-byte-wide) bus. The number of CPU cycles for a B-bytes write access on this bus is

$10+5\left\lceil \frac{B}{8}-1 \right\rceil$, where the square brackets represent the "ceiling" function. For example, an 8-byte write would take

$10+5\left\lceil \frac{8}{8}-1 \right\rceil = 10$ cycles, whereas using the same formula a 12-byte write would take 15 cycles.

Answer the following questions while referring to the C code snippet below:

```
...  #define PORTION 1
     ...
     base = 8*i;
     for (unsigned int j = base; j < base + PORTION; j++)
//assume j is stored in a register
              {
                          data[j] = j;
              }
```

a. [10] <B.1> For a write-through cache, how many CPU cycles are spent on write transfers to the memory for all the combined iterations of the j loop?

b. [10] <B.1> If the cache is configured as a write-back cache, how many CPU cycles are spent on writing back a cache line?

c. [15] <B.1> Change PORTION to 8 and repeat part (a).

d. [15] <B.1> What is the minimum number of array updates to the same cache line (before replacing it) that would render the write-back cache superior?

e. [15] <B.1> Think of a scenario where all the words of the cache line will be written (not necessarily using the above code) and a write-through cache will require fewer total CPU cycles than the write-back cache.

B.5 [10/10/10/10/] <B.2> You are building a system around a processor with in-order execution that runs at 1.1 GHz and has a CPI of 1.35 excluding memory accesses. The only instructions that read or write data from memory are loads (20% of all instructions) and stores (10% of all instructions). The memory system for this computer is composed of a split L1 cache that imposes no penalty on hits. Both the I-cache and D-cache are direct-mapped and hold 32 KB each. The I-cache has a 2% miss rate and 32-byte blocks, and the D-cache is write-through with a 5% miss rate and 16-byte blocks. There is a write buffer on the D-cache that eliminates stalls for 95% of all writes. The 512 KB write-back, unified L2 cache has 64-byte blocks and an access time of 15 ns. It is connected to the L1 cache by a 128-bit data bus that runs at 266 MHz and can transfer one 128-bit word per bus cycle. Of all memory references sent to the L2 cache in this system, 80% are satisfied without going to main memory. Also, 50% of all blocks replaced are dirty. The 128-bit-wide main memory has an access latency of 60 ns, after which any number of bus words may be transferred at the rate of one per cycle on the 128-bit-wide 133 MHz main memory bus.

a. [10] <B.2> What is the average memory access time for instruction accesses?

b. [10] <B.2> What is the average memory access time for data reads?

c. [10] <B.2> What is the average memory access time for data writes?

d. [10] <B.2> What is the overall CPI, including memory accesses?

B.6 [10/15/15] <B.2> Converting miss rate (misses per reference) into misses per instruction relies upon two factors: references per instruction fetched and the fraction of fetched instructions that actually commits.

a. [10] <B.2> The formula for misses per instruction on page B-5 is written first in terms of three factors: miss rate, memory accesses, and instruction count. Each of these factors represents actual events. What is different about writing misses per instruction as *miss rate* times the factor *memory accesses per instruction*?

b. [15] <B.2> Speculative processors will fetch instructions that do not commit. The formula for misses per instruction on page B-5 refers to misses per instruction on the execution path; that is, only the instructions that must actually be executed to carry out the program. Convert the formula for misses per instruction on page B-5 into one that uses only miss rate, references per instruction fetched, and fraction of fetched instructions that commit. Why rely upon these factors rather than those in the formula on page B-5?

c. [15] <B.2> The conversion in part (b) could yield an incorrect value to the extent that the value of the factor references per instruction fetched is not equal to the number of references for any particular instruction. Rewrite the formula of part (b) to correct this deficiency.

B.7 [20] <B.1, B.3> In systems with a write-through L1 cache backed by a write-back L2 cache instead of main memory, a merging write buffer can be simplified. Explain how this can be done. Are there situations where having a full write buffer (instead of the simple version you have just proposed) could be helpful?

B.8 [5/5/5] <B.3> We want to observe the following calculation

$$d_i = a_i + b_i * c_i, \qquad i:(0:511)$$

Arrays *a*, *b*, *c*, and *d* memory layout is displayed below (each has 512 4-byte-wide integer elements).

The above calculation employs a for loop that runs through 512 iterations.

Assume a 32 Kbyte 4-way set associative cache with a single cycle access time. The miss penalty is 100 CPU cycles/access, and so is the cost of a write-back. The cache is a write-back on hits write-allocate on misses cache (Figure B.32).

a. [5] <B3> How many cycles will an iteration take if all three loads and single store miss in the data cache?

b. [5] <B3> If the cache line size is 16 bytes, what is the average number of cycles an average iteration will take? (Hint: Spatial locality!)

c. [5] <B3> If the cache line size is 64 bytes, what is the average number of cycles an average iteration will take?

d. If the cache is direct-mapped and its size is reduced to 2048 bytes, what is the average number of cycles an average iteration will take?

B.9 [20] <B.3> Increasing a cache's associativity (with all other parameters kept constant) statistically reduces the miss rate. However, there can be pathological cases where increasing a cache's associativity would increase the miss rate for a particular workload.

Consider the case of direct-mapped compared to a two-way set associative cache of equal size. Assume that the set associative cache uses the LRU replacement policy. To simplify, assume that the block size is one word. Now, construct a trace of word accesses that would produce more misses in the two-way associative cache.

(Hint: Focus on constructing a trace of accesses that are exclusively directed to a single set of the two-way set associative cache, such that the same trace would exclusively access two blocks in the direct-mapped cache.)

B.10 [10/10/15] <B.3> Consider a two-level memory hierarchy made of L1 and L2 data caches. Assume that both caches use write-back policy on write hit and both have the same block size. List the actions taken in response to the following events:

a. [10] <B.3> An L1 cache miss when the caches are organized in an inclusive hierarchy.

Mem. address in bytes	**Contents**
0–2047	Array *a*
2048–4095	Array *b*
4096–6143	Array *c*
6144–8191	Array *d*

Figure B.32 Arrays layout in memory.

b. [10] <B.3> An L1 cache miss when the caches are organized in an exclusive hierarchy.

c. [15] <B.3> In both parts (a) and (b), consider the possibility that the evicted line might be clean or dirty.

B.11 [15/20] <B.2, B.3> excluding some instructions from entering the cache can reduce conflict misses.

a. [15] <B.3> Sketch a program hierarchy where parts of the program would be better excluded from entering the instruction cache. (Hint: Consider a program with code blocks that are placed in deeper loop nests than other blocks.)

b. [20] <B.2, B.3> Suggest software or hardware techniques to enforce exclusion of certain blocks from the instruction cache.

B.12 [5/15] <B.3> Whereas larger caches have lower miss rates, they also tend to have longer hit times.

Assume a direct-mapped 8 KB cache has 0.22 ns hit time and miss rate m1; also assume a 4-way associative 64 KB cache has 0.52 ns hit time and a miss rate m2.

a. [5] <B.3> If the miss penalty is 100 ns, when would it be advantageous to use the smaller cache to reduce the overall memory access time?

b. [15] <B.3> Repeat part (a) for miss penalties of 10 and 1000 cycles. Conclude when it might be advantageous to use a smaller cache.

B.13 [15] <B.4> A program is running on a computer with a four-entry fully associative (micro) translation lookaside buffer (TLB) (Figure B.33):

The following is a trace of virtual page numbers accessed by a program. For each access indicate whether it produces a TLB hit/miss and, if it accesses the page table, whether it produces a page hit or fault. Put an X under the page table column if it is not accessed (Figures B.34 and B.35).

B.14 [15/15/15/15/] <B.4> Some memory systems handle TLB misses in software (as an exception), while others use hardware for TLB misses.

a. [15] <B.4> What are the trade-offs between these two methods for handling TLB misses?

b. [15] <B.4> Will TLB miss handling in software always be slower than TLB miss handling in hardware? Explain.

VP#	PP#	Entry valid
5	30	1
7	1	0
10	10	1
15	25	1

Figure B.33 TLB contents (problem B.12).

Virtual page index	Physical page #	Present
0	3	Y
1	7	N
2	6	N
3	5	Y
4	14	Y
5	30	Y
6	26	Y
7	11	Y
8	13	N
9	18	N
10	10	Y
11	56	Y
12	110	Y
13	33	Y
14	12	N
15	25	Y

Figure B.34 Page table contents.

Virtual page accessed	TLB (hit or miss)	Page table (hit or fault)
1		
5		
9		
14		
10		
6		
15		
12		
7		
2		

Figure B.35 Page access trace.

c. [15] <B.4> Are there page table structures that would be difficult to handle in hardware but possible in software? Are there any such structures that would be difficult for software to handle but easy for hardware to manage?

d. [15] <B.4> Why are TLB miss rates for floating-point programs generally higher than those for integer programs?

B.15 [20/20] <B.5> It is possible to provide more flexible protection than that in the Intel Pentium architecture by using a protection scheme similar to that used in the Hewlett-Packard Precision Architecture (HP/PA). In such a scheme, each page table entry contains a "protection ID" (key) along with access rights for the page. On each reference, the CPU compares the protection ID in the page table entry with those stored in each of four protection ID registers (access to these registers requires that the CPU be in supervisor mode). If there is no match for the protection ID in the page table entry or if the access is not a permitted access (writing to a read-only page, for example), an exception is generated.

a. [20] <B.5> Explain how this model could be used to facilitate the construction of operating systems from relatively small pieces of code that cannot overwrite each other (microkernels). What advantages might such an operating system have over a monolithic operating system in which any code in the OS can write to any memory location?

b. [20] <B.5> A simple design change to this system would allow two protection IDs for each page table entry, one for read access and the other for either write or execute access (the field is unused if neither the writable nor executable bit is set). What advantages might there be from having different protection IDs for read and write capabilities? (*Hint*: Could this make it easier to share data and code between processes?)

C.1 Introduction C-2
C.2 The Major Hurdle of Pipelining—Pipeline Hazards C-10
C.3 How Is Pipelining Implemented? C-26
C.4 What Makes Pipelining Hard to Implement? C-37
C.5 Extending the RISC V Integer Pipeline to Handle Multicycle Operations C-45
C.6 Putting It All Together: The MIPS R4000 Pipeline C-55
C.7 Cross-Cutting Issues C-65
C.8 Fallacies and Pitfalls C-70
C.9 Concluding Remarks C-71
C.10 Historical Perspective and References C-71
Updated Exercises by Diana Franklin C-71

C

Pipelining: Basic and Intermediate Concepts

> It is quite a three-pipe problem.
>
> **Sir Arthur Conan Doyle,**
> *The Adventures of Sherlock Holmes*

C.1 Introduction

Many readers of this text will have covered the basics of pipelining in another text (such as our more basic text *Computer Organization and Design*) or in another course. Because Chapter 3 builds heavily on this material, readers should ensure that they are familiar with the concepts discussed in this appendix before proceeding. As you read Chapter 3, you may find it helpful to turn to this material for a quick review.

We begin the appendix with the basics of pipelining, including discussing the data path implications, introducing hazards, and examining the performance of pipelines. This section describes the basic five-stage RISC pipeline that is the basis for the rest of the appendix. Section C.2 describes the issue of hazards, why they cause performance problems, and how they can be dealt with. Section C.3 discusses how the simple five-stage pipeline is actually implemented, focusing on control and how hazards are dealt with.

Section C.4 discusses the interaction between pipelining and various aspects of instruction set design, including discussing the important topic of exceptions and their interaction with pipelining. Readers unfamiliar with the concepts of precise and imprecise interrupts and resumption after exceptions will find this material useful, because they are key to understanding the more advanced approaches in Chapter 3.

Section C.5 discusses how the five-stage pipeline can be extended to handle longer-running floating-point instructions. Section C.6 puts these concepts together in a case study of a deeply pipelined processor, the MIPS R4000/4400, including both the eight-stage integer pipeline and the floating-point pipeline. The MIPS R40000 is similar to a single-issue embedded processor, such as the ARM Cortex-A5, which became available in 2010, and was used in several smart phones and tablets.

Section C.7 introduces the concept of dynamic scheduling and the use of scoreboards to implement dynamic scheduling. It is introduced as a cross-cutting issue, because it can be used to serve as an introduction to the core concepts in Chapter 3, which focused on dynamically scheduled approaches. Section C.7 is also a gentle introduction to the more complex Tomasulo's algorithm covered in Chapter 3. Although Tomasulo's algorithm can be covered and understood without introducing scoreboarding, the scoreboarding approach is simpler and easier to comprehend.

What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast processors, and even processors that cost less than a dollar are pipelined.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

In an automobile assembly line, *throughput* is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving an instruction one step down the pipeline is a *processor cycle*. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line the longest step would determine the time between advancing cars in the line. In a computer, this processor cycle is almost always 1 clock cycle.

The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined processor will not have its minimum possible value, yet it can be close.

Pipelining yields a reduction in the average execution time per instruction. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining reduces the CPI. This is the primary view we will take.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapter 4), it is not visible to the programmer.

The Basics of the RISC V Instruction Set

Throughout this book we use RISC V, a load-store architecture, to illustrate the basic concepts. Nearly all the ideas we introduce in this book are applicable to other processors, but the implementation may be much more complicated with complex instructions. In this section, we make use of the core of the RISC V architecture; see Chapter 1 for a full description. Although we use RISC V, the concepts are significantly similar in that they will apply to any RISC, including the core architectures of ARM and MIPS. All RISC architectures are characterized by a few key properties:

- All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
- The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively. Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- The instruction formats are few in number, with all instructions typically being one size. In RISC V, the register specifiers: rs1, rs2, and rd are always in the same place simplifying the control.

These simple properties lead to dramatic simplifications in the implementation of pipelining, which is why these instruction sets were designed this way. Chapter 1 contains a full description of the RISC V ISA, and we assume the reader has read Chapter 1.

A Simple Implementation of a RISC Instruction Set

To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented *without* pipelining. This section shows a simple implementation where every instruction takes at most 5 clock cycles. We will extend this basic implementation to a pipelined version, resulting in a much lower CPI. Our unpipelined implementation is not the most economical or the highest-performance implementation without pipelining. Instead, it is designed to lead naturally to a pipelined implementation. Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; these are introduced in this section to simplify pipelining. Our implementation will focus only on a pipeline for an integer subset of a RISC architecture that consists of load-store word, branch, and integer ALU operations.

Every instruction in this RISC subset can be implemented in, at most, 5 clock cycles. The 5 clock cycles are as follows.

1. *Instruction fetch cycle* (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential instruction by adding 4 (because each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle* (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC.

   Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture. This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) For loads and ALU immediate operations, the immediate field is always in the same place, so we can easily sign extend it. (For a more complete implementation of RISC V, we would need to compute two different sign-extended values, because the immediate field for store is in a different location.)

3. *Execution/effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of three functions, depending on the instruction type.

   - Memory reference—The ALU adds the base register and the offset to form the effective address.
   - Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
   - Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.
   - Conditional branch—Determine whether the condition is true.

   In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, because no instruction needs to simultaneously calculate a data address and perform an operation on the data.

4. *Memory access* (MEM):

   If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. *Write-back cycle* (WB):

   - Register-Register ALU instruction or load instruction:

   Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

In this implementation, branch instructions require three cycles, store instructions require four cycles, and all other instructions require five cycles. Assuming a branch frequency of 12% and a store frequency of 10%, a typical instruction distribution leads to an overall CPI of 4.66. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level; we leave the improvement of this design as an exercise for you and instead focus on pipelining this version.

The Classic Five-Stage Pipeline for a RISC Processor

We can pipeline the execution described in the previous section with almost no changes by simply starting a new instruction on each clock cycle. (See why we chose this design?) Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline. This results in the execution pattern shown in Figure C.1, which is the typical way a pipeline structure is drawn. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

You may find it hard to believe that pipelining is as simple as this; it's not. In this and the following sections, we will make our RISC pipeline "real" by dealing with problems that pipelining introduces.

To start with, we have to determine what happens on every clock cycle of the processor and make sure we don't try to perform two different operations with the same data path resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy. Figure C.2 shows a simplified version of a RISC data path drawn in pipeline fashion. As you can see, the major functional units are used in different cycles, and hence overlapping the execution of multiple

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

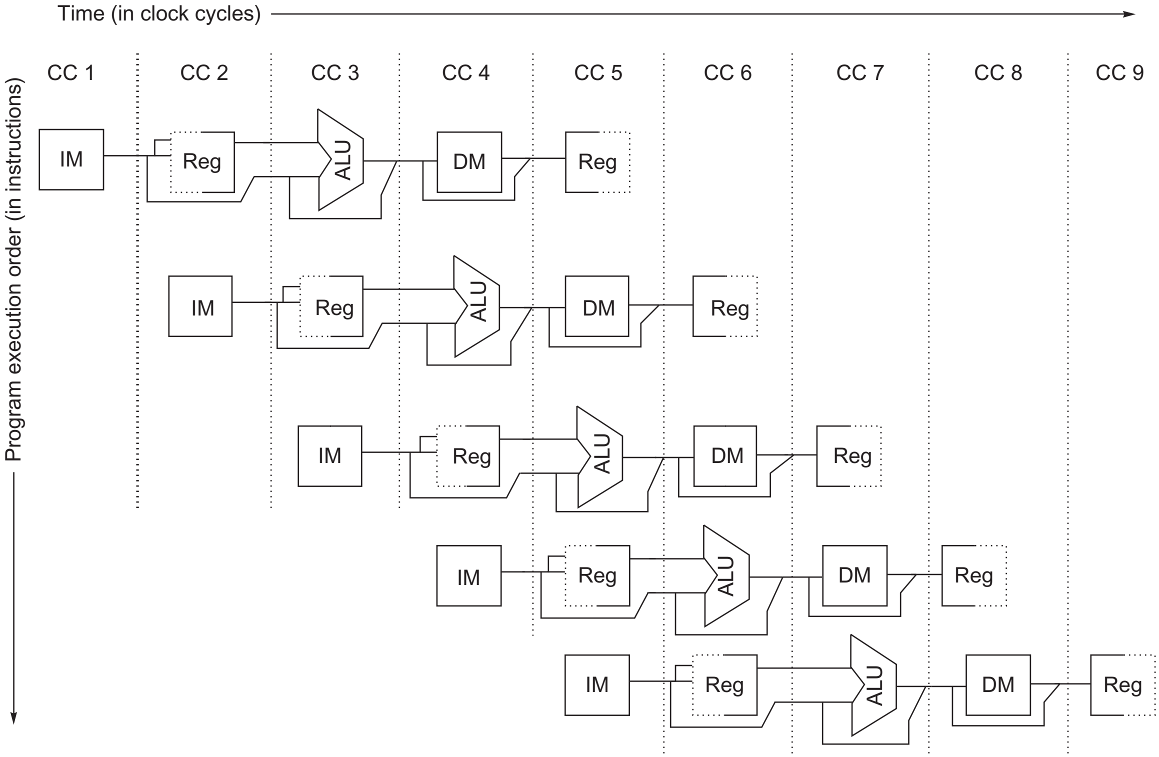


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This figure shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

instructions introduces relatively few conflicts. There are three observations on which this fact rests.

First, we use separate instruction and data memories, which we would typically implement with separate instruction and data caches (discussed in Chapter 2). The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

Second, the register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we simply show the register file in two places. Hence, we need to perform two reads and one write every clock cycle.

To handle reads and a write to the same register (and for another reason, which will become obvious shortly), we perform the register write in the first half of the clock cycle and the read in the second half.

Third, Figure C.2 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target address during ID. One further problem is that we need the ALU in the ALU stage to evaluate the branch condition. Actually, we don't really need a full ALU to evaluate the comparison between two registers, but we need enough of the function that it has to occur in this pipestage.

Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. Figure C.3 shows the pipeline drawn with these pipeline registers.

Although many figures will omit such registers for simplicity, they are required to make the pipeline operate properly and must be present. Of course, similar registers would be needed even in a multicycle data path that had no pipelining (because only values in registers are preserved across clock boundaries). In the case of a pipelined processor, the pipeline registers also play the key role of carrying intermediate results from one stage to another where the source and destination may not be directly adjacent. For example, the register value to be stored during a store instruction is read during ID, but not actually used until MEM; it is passed through two pipeline registers to reach the data memory during the MEM stage. Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers. It is sometimes useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect, so the registers are called IF/ID, ID/EX, EX/MEM, and MEM/WB.

Basic Performance Issues in Pipelining

Pipelining increases the processor instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline, as we will see in the next section. In addition to limitations arising from pipeline latency, limits arise from imbalance

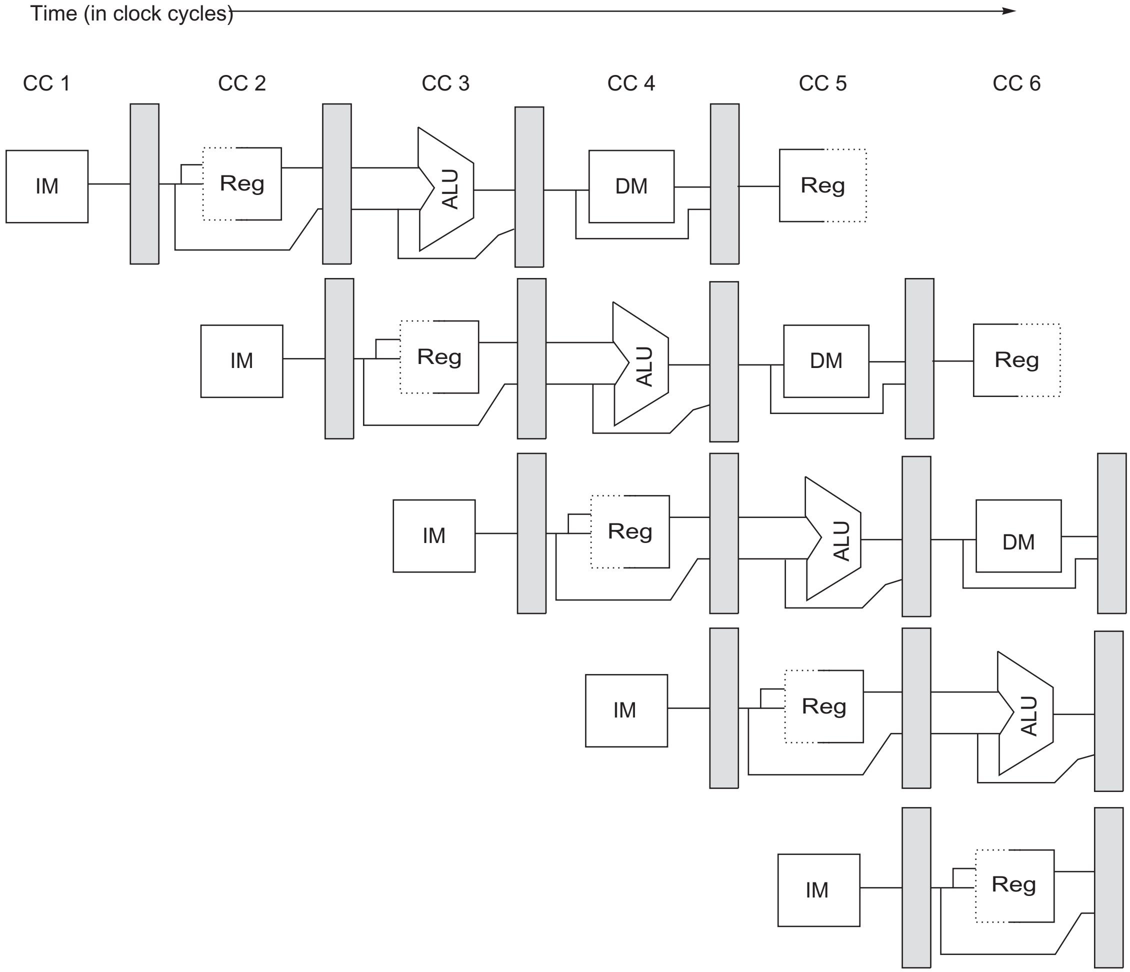


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance because the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle. Clock skew, which is the maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, because there is no time left in the cycle for useful work. The interested reader should see Kunkel and Smith (1986).

Example Consider the unpipelined processor in the previous section. Assume that it has a 4 GHz clock (or a 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.1 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI}\\ &= 0.5\,\text{ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 0.5\ \text{ns} \times 4.4\\ &= 2.2\ \text{ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 0.5 + 0.1 or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{2.2\ \text{ns}}{0.6\ \text{ns}} = 3.7\ \text{times}\end{aligned}$$

The 0.1 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's Law tells us that the overhead limits the speedup.

This simple RISC pipeline would function just fine for integer instructions if every instruction were independent of every other instruction in the pipeline. In reality, instructions in the pipeline can depend on one another; this is the topic of the next section.

C.2 The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution. In modern processors, structural hazards occur primarily in special purpose functional units that are less frequently used (such as floating point divide or other complex long running instructions). They are not a major performance factor, assuming programmers and compiler writers are aware of the lower throughput of these instructions. Instead of spending more time on this infrequent case, we focus on the two other hazards that are much more frequent.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to *stall* the pipeline. Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed. For the pipelines we discuss in this appendix, when an instruction is stalled, all instructions issued *later* than the stalled instruction—and hence not as far along in the pipeline—are also stalled. Instructions issued *earlier* than the stalled instruction—and hence farther along in the pipeline—must continue, because otherwise the hazard will never clear. As a result, no new instructions are fetched during the stall. We will see several examples of how pipeline stalls operate in this section—don't worry, they aren't as complex as they might sound!

Performance of Pipelines With Stalls

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}\\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}\end{aligned}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Because it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipelines stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume that the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpiplined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Assume instruction i occurs in program order before instruction j and both instructions use register x, then there are three different types of hazards that can occur between i and j:

1. Read After Write (RAW) hazard: the most common, these occur when a read of register x by instruction j occurs before the write of register x by instruction i. If this hazard were not prevented instruction j would use the wrong value of x.
2. Write After Read (WAR) hazard: this hazard occurs when read of register x by instruction i occurs after a write of register x by instruction j. In this case, instruction i would use the wrong value of x. WAR hazards are impossible in the simple five stage, integrer pipeline, but they occur when instructions are reordered, as we will see when we discuss dynamically scheduled pipelines beginning on page C.65.
3. Write After Write (WAW) hazard: this hazard occurs when write of register x by instruction i occurs after a write of register x by instruction j. When this occurs, register x will have the wrong value going forward. WAR hazards are also impossible in the simple five stage, integrer pipeline, but they occur when instructions are reordered or when running times vary, as we will see later.

Chapter 3 explores the issues of data dependence and hazards in much more detail. For now, we focus only on RAW hazards.

Consider the pipelined execution of these instructions:

```
add     x1,x2,x3
sub     x4,x1,x5
and     x6,x1,x7
or      x8,x1,x9
xor     x10,x1,x11
```

All the instructions after the `add` use the result of the `add` instruction. As shown in Figure C.4, the `add` instruction writes the value of x1 in the WB pipe stage, but the `sub` instruction reads the value during its ID stage, which results in a RAW hazard. Unless precautions are taken to prevent it, the `sub` instruction will read the wrong value and try to use it. In fact, the value used by the `sub` instruction is not even deterministic: though we might think it logical to assume that `sub` would always use the value of x1 that was assigned by an instruction prior to `add`, this is not

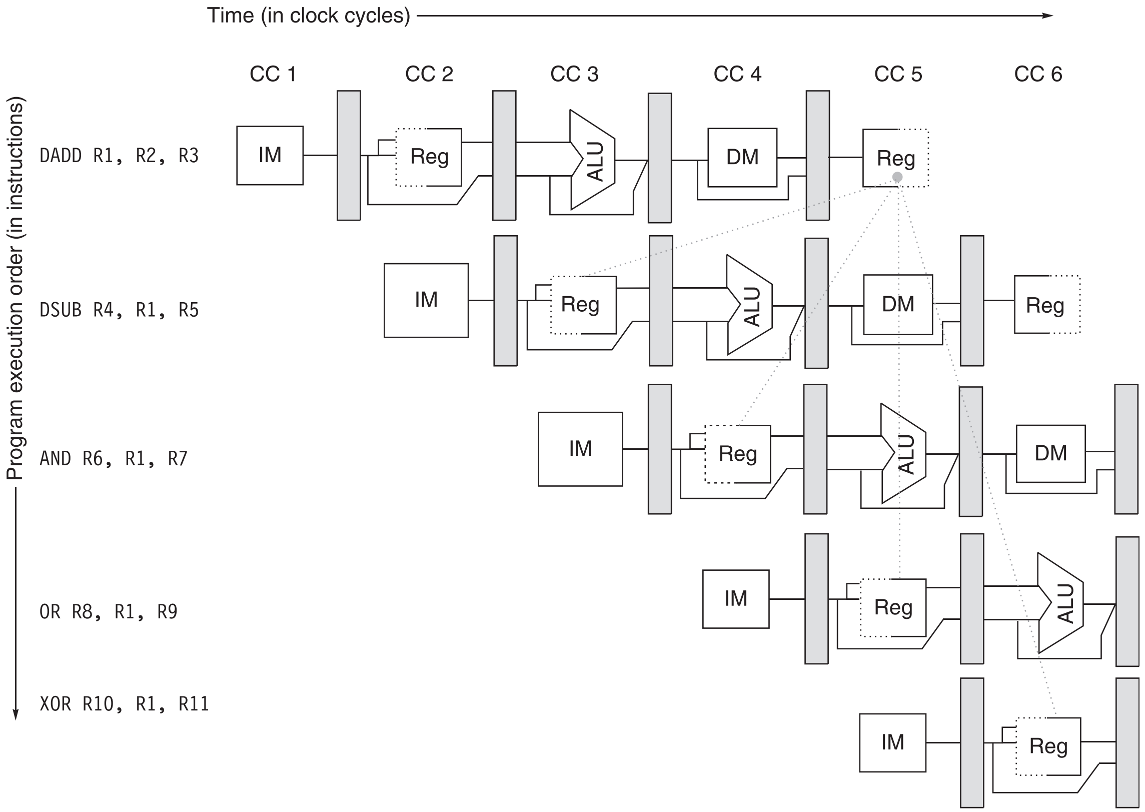

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

Figure C.4 The use of the result of the `add` instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

always the case. If an interrupt should occur between the `add` and `sub` instructions, the WB stage of the `add` will complete, and the value of x1 at that point will be the result of the `add`. This unpredictable behavior is obviously unacceptable.

The `and` instruction also creates a possible RAW hazard. As we can see from Figure C.4, the write of x1 does not complete until the end of clock cycle 5. Thus, the `and` instruction that reads the registers during clock cycle 4 will receive the wrong results.

The `xor` instruction operates properly because its register read occurs in clock cycle 6, after the register write. The `or` instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half. Note that the `xor` instruction still depends on the `add`, but it no longer creates a hazard; a topic we explore in more detail in Chapter 3.

The next subsection discusses a technique to eliminate the stalls for the hazard involving the `sub` and `and` instructions.

Minimizing Data Hazard Stalls by Forwarding

The problem posed in Figure C.4 can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*). The key insight in forwarding is that the result is not really needed by the `sub` until after the `add` actually produces it. If the result can be moved from the pipeline register where the `add` stores it to where the `sub` needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Notice that with forwarding, if the `sub` is stalled, the `add` will be completed and the bypass will not be activated. This relationship is also true for the case of an interrupt between the two instructions.

As the example in Figure C.4 shows, we need to forward results not only from the immediately previous instruction but also possibly from an instruction that started two cycles earlier. Figure C.5 shows our example with the bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

Forwarding can be generalized to include passing a result directly to the functional unit that requires it: a result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from

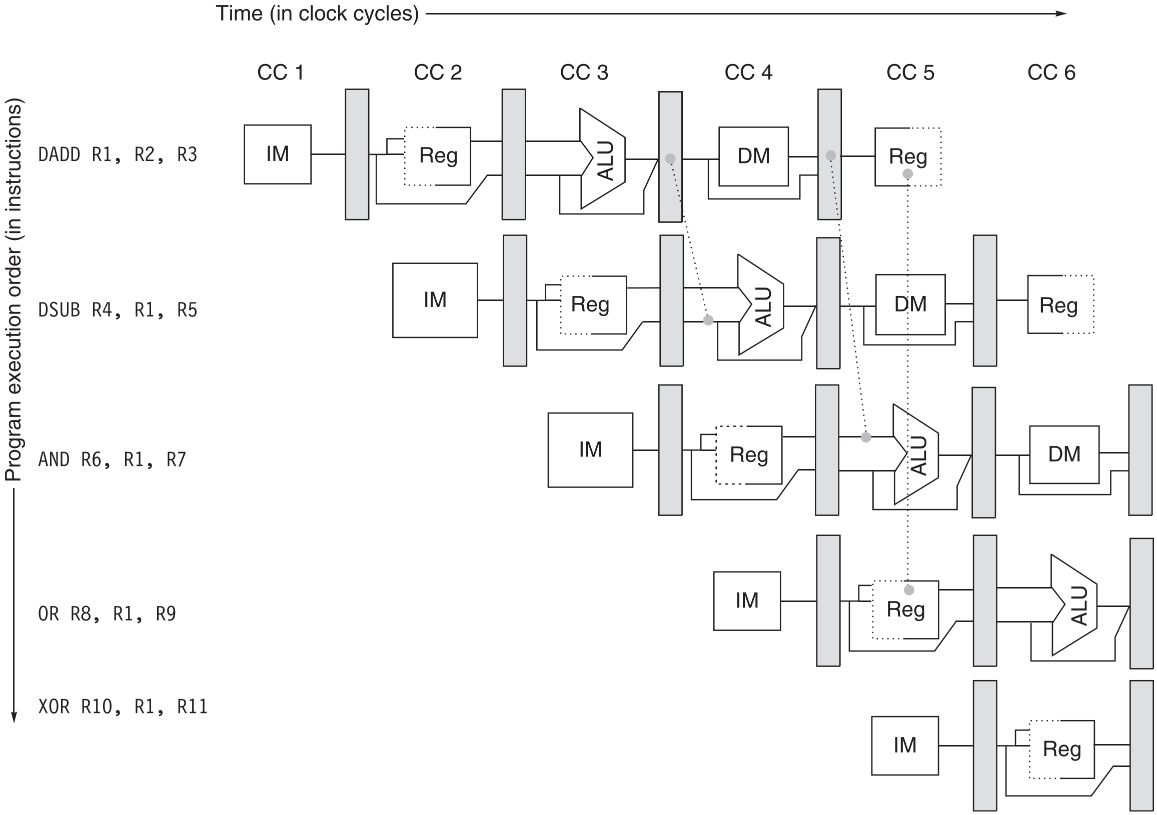

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
Program execution order (in instructions)
DADD R1, R2, R3
DSUB R4, R1, R5
AND R6, R1, R7
OR R8, R1, R9
XOR R10, R1, R11
IM
Reg
ALU
DM
Reg

Figure C.5 A set of instructions that depends on the `add` result uses forwarding paths to avoid the data hazard. The inputs for the `sub` and `and` instructions forward from the pipeline registers to the first ALU input. The `or` receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the `and` instruction was `and x6,x1,x4`.

the result of a unit to the input of the same unit. Take, for example, the following sequence:

```
add     x1,x2,x3
ld      x4,0(x1)
sd      x4,12(x1)
```

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs. Figure C.6 shows all the forwarding paths for this example.

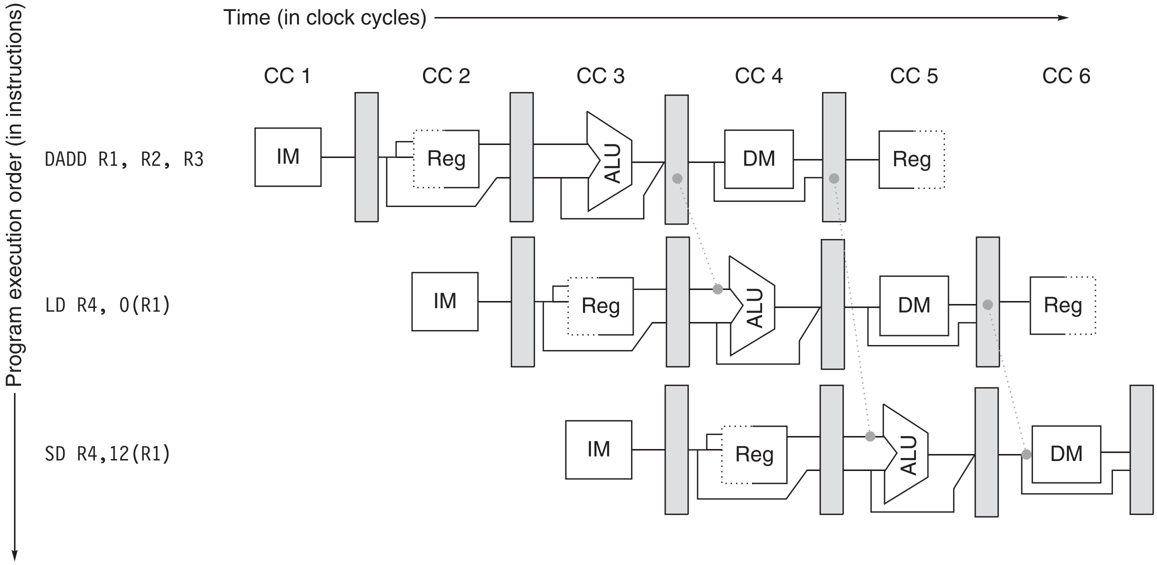


Figure C.6 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
ld      x1,0(x2)
sub     x4,x1,x5
and     x6,x1,x7
or      x8,x1,x9
```

The pipelined data path with the bypass paths for this example is shown in Figure C.7. This case is different from the situation with back-to-back ALU operations. The `ld` instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the `sub` instruction needs to have the data by the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware. As Figure C.7 shows, such a forwarding path would have to operate backward in time—a capability not yet available to computer designers! We *can* forward the result immediately to the ALU from the pipeline registers for use in the `and` operation, which begins 2 clock cycles after the load. Likewise, the `or` instruction has no problem, because it receives the value through the register file. For the `sub` instruction, the forwarded

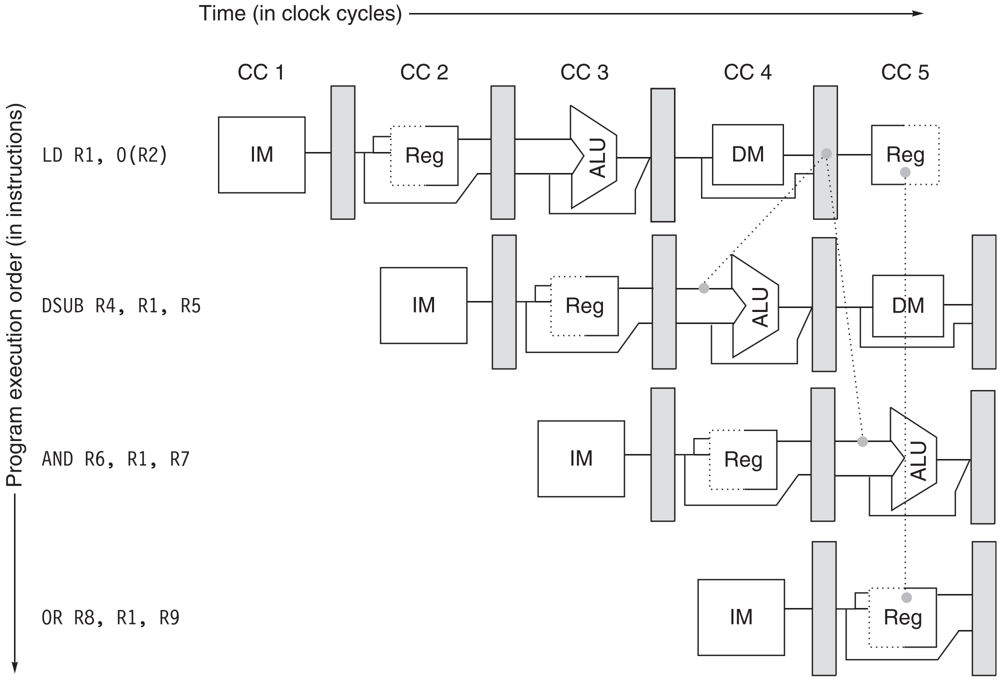


Figure C.7 The load instruction can bypass its results to the `and` and `or` instructions, but not to the `sub`, because that would mean forwarding the result in "negative time."

result arrives too late—at the end of a clock cycle, when it is needed at the beginning.

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it. This pipeline interlock introduces a stall or bubble, just as it did for the structural hazard. The CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case).

Figure C.8 shows the pipeline before and after the stall using the names of the pipeline stages. Because the stall causes the instructions starting with the `sub` to move one cycle later in time, the forwarding to the `and` instruction now goes through the register file, and no forwarding at all is needed for the `or` instruction. The insertion of the bubble causes the number of cycles to complete this sequence to grow by one. No instruction is started during clock cycle 4 (and none finishes during cycle 6).

`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	EX	MEM	WB			
`and x6,x1,x7`			IF	ID	EX	MEM	WB		
`or x8,x1,x9`				IF	ID	EX	MEM	WB	
`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	Stall	EX	MEM	WB		
`and x6,x1,x7`			IF	Stall	ID	EX	MEM	WB	
`or x8,x1,x9`				Stall	IF	ID	EX	MEM	WB

Figure C.8 In the top half, we can see why a stall is needed: the MEM cycle of the load produces a value that is needed in the EX cycle of the `sub`, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch Hazards

Control hazards can cause a greater performance loss for our RISC V pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken*, or *untaken*. If instruction *i* is a taken branch, then the PC is usually not changed until the end of ID, after the completion of the address calculation and comparison.

Figure C.9 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary because the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly.

One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Figure C.9 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four simple compile time schemes in this subsection. In these four schemes the actions for a branch are static—they are fixed for each branch during the entire execution. The software can try to minimize the branch penalty using knowledge of the hardware scheme and of branch behavior. We will then look at hardware-based schemes that dynamically predict branch behavior, and Chapter 3 looks at more powerful hardware techniques for dynamic branch prediction.

The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure C.9. In this case, the branch penalty is fixed and cannot be reduced by software.

A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Here, care must be taken not to change the processor state until the branch outcome is definitely known. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to "back out" such a change.

In the simple five-stage pipeline, this *predicted-not-taken* or *predicted-untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address. Figure C.10 shows both situations.

An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

begin fetching and executing at the target. This buys us a one-cycle improvement when the branch is actually taken, because we know the target address at the end of ID, one cycle before we know whether the branch condition is satisfied in the ALU stage. In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice.

A fourth scheme, which was heavily used in early RISC processors is called *delayed branch*. In a delayed branch, the execution cycle with a branch delay of one is

branch instruction
sequential successor$_1$
branch target if taken

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in Figure C.11. Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty.The job of the compiler is to make the successor instructions valid and useful.

Although the delayed branch was useful for short simple pipelines at a time when hardware prediction was too expensive, the technique complicates implementation when there is dynamic branch prediction. For this reason, RISC V appropriately omitted delayed branches.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.11 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there was only one delay slot for most RISC architectures that incorporated them) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: if the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

Performance of Branch Schemes

What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

we obtain:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches. However, the latter dominate because they are more frequent.

Example For a deeper pipeline, such as that in a MIPS R4000 and later RISC processors, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.12.

Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Answer We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure C.13.

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.12 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

	Additions to the CPI from branch costs			
Branch scheme	**Unconditional branches**	**Untaken conditional branches**	**Taken conditional branches**	**All branches**
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Figure C.13 CPI penalties for three branch-prediction schemes and a deeper pipeline.

The differences among the schemes are substantially increased with this longer delay. If the base CPI were 1 and branches were the only source of stalls, the ideal pipeline would be 1.56 times faster than a pipeline that used the stall-pipeline scheme. The predicted-untaken scheme would be 1.13 times better than the stall-pipeline scheme under the same assumptions.

Reducing the Cost of Branches Through Prediction

As pipelines get deeper and the potential penalty of branches increases, using delayed branches and similar schemes becomes insufficient. Instead, we need to turn to more aggressive means for predicting branches. Such schemes fall into two classes: low-cost static schemes that rely on information available at compile time and strategies that predict branches dynamically based on program behavior. We discuss both approaches here.

Static Branch Prediction

A key way to improve compile-time branch prediction is to use profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure C.14 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%. The fact that the misprediction rate for the integer programs is higher and such programs typically have a higher branch frequency is a major limitation for static branch prediction. In the next section, we consider dynamic branch predictors, which most recent processors have employed.

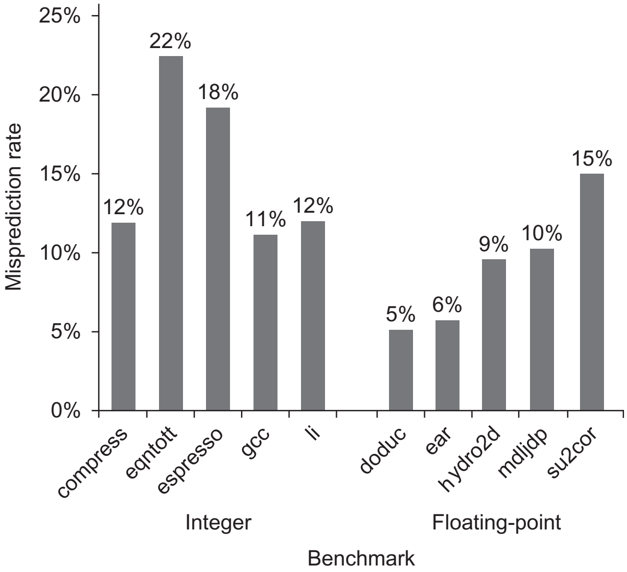


Figure C.14 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

Dynamic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs.

With such a buffer, we don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch-prediction scheme.

This simple 1-bit prediction scheme has a performance shortcoming: even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, because the misprediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure C.15 shows the finite-state processor for a 2-bit prediction scheme.

A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. As Figure C.15 shows, if the prediction turns out to be wrong, the prediction bits are changed.

What kind of accuracy can be expected from a branch-prediction buffer using 2 bits per entry on real applications? Figure C.16 shows that for the SPEC89 benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy

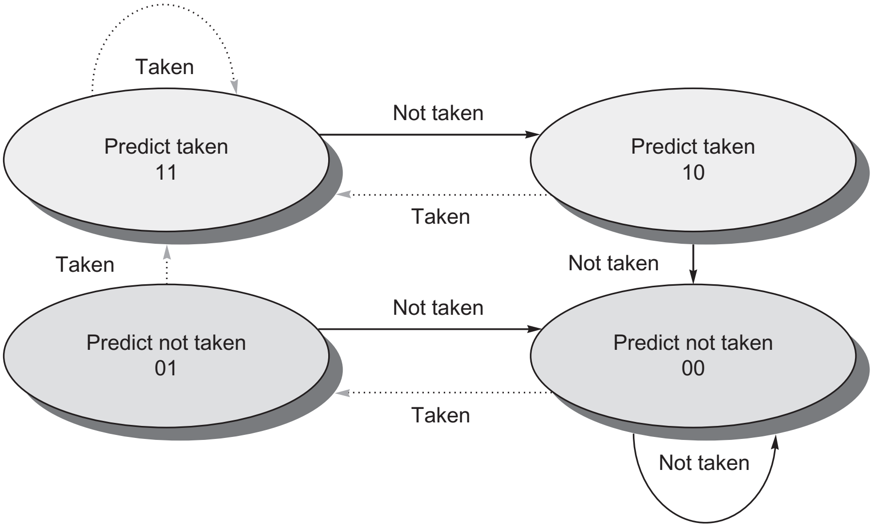


Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

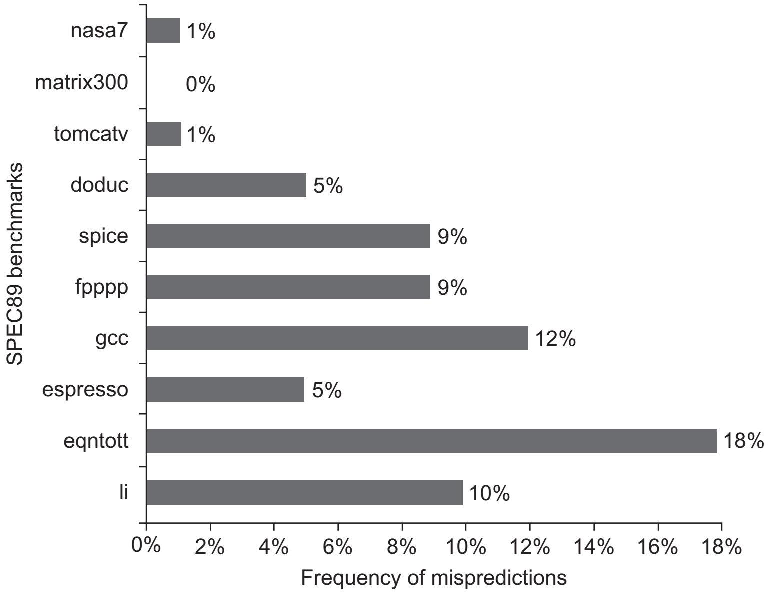


Figure C.16 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. (1992). Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

ranging from over 99% to 82%, or a *misprediction rate* of 1%–18%. A 4K entry buffer, like that used for these results, is considered small in 2017, and a larger buffer could produce somewhat better results.

As we try to exploit more ILP, the accuracy of our branch prediction becomes critical. As we can see in Figure C.16, the accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs. We can attack this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction. A buffer with 4K entries, however, as Figure C.17 shows, performs quite comparably to an infinite buffer, at least for benchmarks like those in SPEC. The data in Figure C.17 make it clear that the hit rate of the buffer is not the major limiting factor. As we mentioned, simply increasing the number of bits per predictor without changing the predictor structure also has little impact. Instead, we need to look at how we might increase the accuracy of each predictor, as we will in Chapter 3.

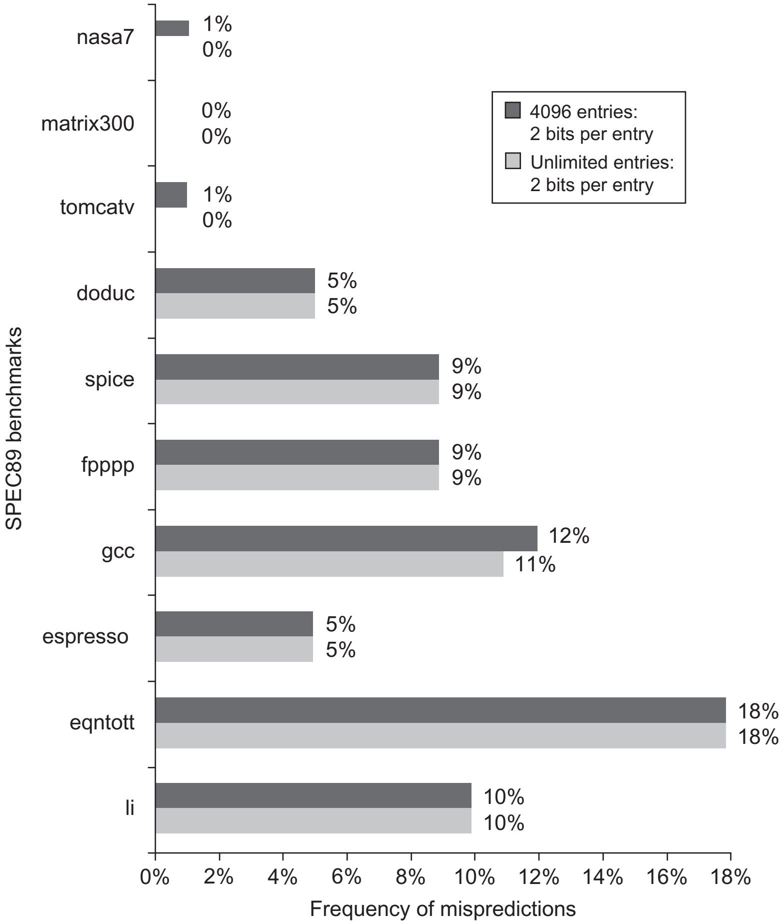


Figure C.17 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although these data are for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

C.3 How Is Pipelining Implemented?

Before we proceed to basic pipelining, we need to review a simple implementation of an unpipelined version of RISC V.

A Simple Implementation of RISC V

In this section we follow the style of Section C.1, showing first a simple unpipelined implementation and then the pipelined implementation. This time, however, our example is specific to the RISC V architecture.

In this subsection, we focus on a pipeline for an integer subset of RISC V that consists of load-store word, branch equal, and integer ALU operations. Later in this appendix we will incorporate the basic floating-point operations. Although we discuss only a subset of RISC V, the basic principles can be extended to handle all the instructions; for example, adding store involves some additional computing of the immediate field. We initially used a less aggressive implementation of a branch instruction. We show how to implement the more aggressive version at the end of this section.

Every RISC V instruction can be implemented in, at most, 5 clock cycles. The 5 clock cycles are as follows:

1. *Instruction fetch cycle* (IF):

   ```
   IR ← Mem[PC];
   NPC ← PC + 4;
   ```

   Operation—Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; likewise, the register NPC is used to hold the next sequential PC.

2. *Instruction decode/register fetch cycle* (ID):

   ```
   A ← Regs[rs1];
   B ← Regs[rs2];
   Imm ← sign-extended immediate field of IR;
   ```

   Operation—Decode the instruction and access the register file to read the registers (rs1 and rs2 are the register specifiers). The outputs of the general-purpose registers are read into two temporary registers (A and B) for use in later clock cycles. The lower 16 bits of the IR are also sign extended and stored into the temporary register Imm, for use in the next cycle.

   Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the RISC V instruction format. Because the immediate portion of a load and an ALU immediate is located in an identical place in every RISC V instruction, the sign-extended immediate is also calculated during this cycle in case it is needed in the next cycle. For stores, a separate sign-extension is needed, because the immediate field is split in two pieces.

3. *Execution/effective address cycle* (EX):

   The ALU operates on the operands prepared in the prior cycle, performing one of four functions depending on the RISC V instruction type:

   - Memory reference:

     ```
     ALUOutput ← A + Imm;
     ```

   Operation—The ALU adds the operands to form the effective address and places the result into the register ALUOutput.

- Register-register ALU instruction:

```
ALUOutput ← A func B;
```

Operation—The ALU performs the operation specified by the function code (a combination of the func3 and func7 fields) on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.

- Register-Immediate ALU instruction:

```
ALUOutput ← A op Imm;
```

Operation—The ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. The result is placed in the temporary register ALUOutput.

- Branch:

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == B)
```

Operation—The ALU adds the NPC to the sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute the address of the branch target. Register A, which has been read in the prior cycle, is checked to determine whether the branch is taken, by comparison with Register B, because we consider only branch equal.

The load-store architecture of RISC V means that effective address and execution cycles can be combined into a single clock cycle, because no instruction needs to simultaneously calculate a data address, calculate an instruction target address, and perform an operation on the data. The other integer instructions not included herein are jumps of various forms, which are similar to branches.

4. *Memory access/branch completion cycle* (MEM):

The PC is updated for all instructions: `PC ← NPC;`

- Memory reference:

```
LMD ← Mem[ALUOutput] or
Mem[ALUOutput] ← B;
```

Operation—Access memory if needed. If the instruction is a load, data return from memory and are placed in the LMD (load memory data) register; if it is a store, then the data from the B register are written into memory. In either case, the address used is the one computed during the prior cycle and stored in the register ALUOutput.

- Branch:

  `if (cond) PC ← ALUOutput`

  Operation—If the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput.

5. *Write-back cycle* (WB):

   - Register-register or Register-immediate ALU instruction:

     `Regs[rd] ← ALUOutput;`

   - Load instruction:

     `Regs[rd] ← LMD;`

   Operation—Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput) with rd designating the register.

Figure C.18 shows how an instruction flows through the data path. At the end of each clock cycle, every value computed during that clock cycle and required on a later clock cycle (whether for this instruction or the next) is written into a storage device, which may be memory, a general-purpose register, the PC, or a temporary register (i.e., LMD, Imm, A, B, IR, NPC, ALUOutput, or Cond). The temporary registers hold values between clock cycles for one instruction, while the other storage elements are visible parts of the state and hold values between successive instructions.

Although all processors today are pipelined, this multicycle implementation is a reasonable approximation of how most processors would have been implemented in earlier times. A simple finite-state machine could be used to implement the control following the five-cycle structure shown herein. For a much more complex processor, microcode control could be used. In either event, an instruction sequence like the one described in this section would determine the structure of the control.

There are some hardware redundancies that could be eliminated in this multicycle implementation. For example, there are two ALUs: one to increment the PC and one used for effective address and ALU computation. Because they are not needed on the same clock cycle, we could merge them by adding additional multiplexers and sharing the same ALU. Likewise, instructions and data could be stored in the same memory, because the data and instruction accesses happen on different clock cycles.

Rather than optimize this simple implementation, we will leave the design as it is in Figure C.18, because this provides us with a better base for the pipelined implementation.

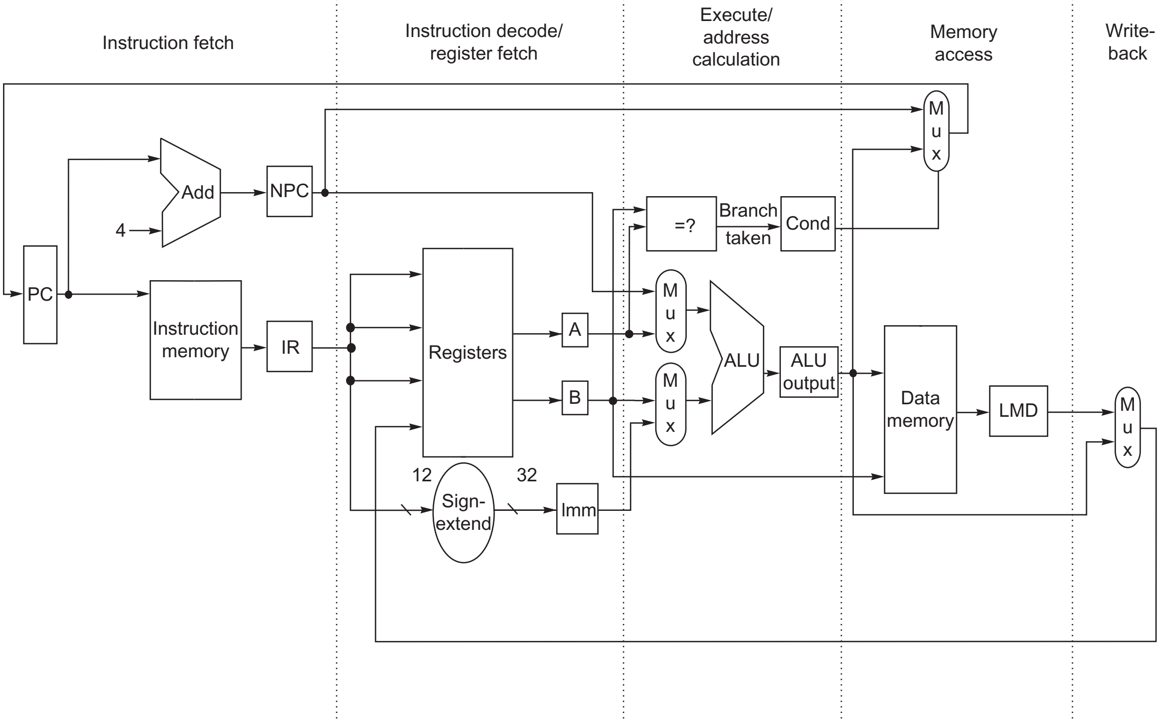

Instruction fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access
Write-back
PC
Add
4
NPC
Instruction memory
IR
Registers
A
B
12
Sign-extend
32
Imm
=?
Branch taken
Cond
Mux
Mux
Mux
ALU
ALU output
Data memory
LMD
Mux

Figure C.18 The implementation of the RISC V data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write-back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, because they indicate the possibility of hazards.

A Basic Pipeline for RISC V

As before, we can pipeline the data path of Figure C.18 with almost no changes by starting a new instruction on each clock cycle. Because every pipe stage is active on every clock cycle, all operations in a pipe stage must complete in 1 clock cycle and any combination of operations must be able to occur at once. Furthermore, pipelining the data path requires that values passed from one pipe stage to the next must be placed in registers. Figure C.19 shows the RISC V pipeline with the appropriate registers, called *pipeline registers* or *pipeline latches*, between each pipeline stage. The registers are labeled with the names of the stages they connect.

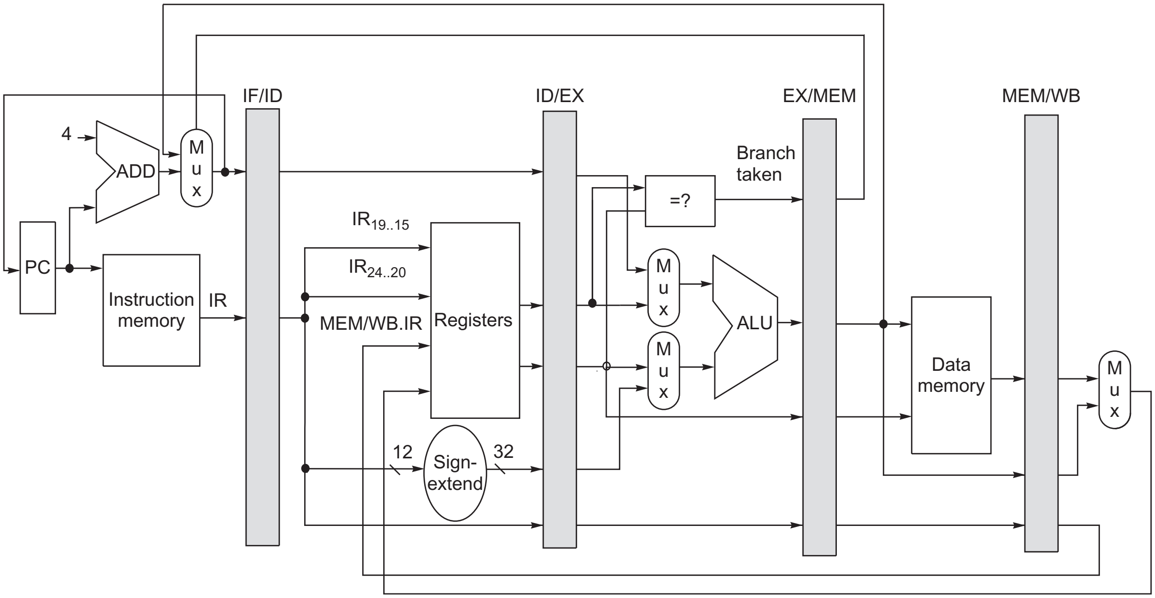


Figure C.19 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence, there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, because two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

Figure C.19 is drawn so that connections through the pipeline registers from one stage to another are clear.

All of the registers needed to hold values temporarily between clock cycles within one instruction are subsumed into these pipeline registers. The fields of the instruction register (IR), which is part of the IF/ID register, are labeled when they are used to supply register names. The pipeline registers carry both data and control from one pipeline stage to the next. Any value needed on a later pipeline stage must be placed in such a register and copied from one pipeline register to the next, until it is no longer needed. If we tried to just use the temporary registers we had in our earlier unpipelined data path, values could be overwritten before all uses were completed. For example, the field of a register operand used for a write on a load or ALU operation is supplied from the MEM/WB pipeline register rather than from the IF/ID register. This is because we want a load or ALU operation to write the register designated by that operation, not the register field of the instruction currently transitioning from IF to ID! This destination register field is simply copied from one pipeline register to the next, until it is needed during the WB stage.

Any instruction is active in exactly one stage of the pipeline at a time; therefore, any actions taken on behalf of an instruction occur between a pair of pipeline registers. Thus, we can also look at the activities of the pipeline by examining what has to happen on any pipeline stage depending on the instruction type. Figure C.20 shows this view. Fields of the pipeline registers are named so as to show the flow

Stage	**Any instruction**		
IF	IF/ID.IR←Mem[PC] IF/ID.NPC,PC←(if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});		
ID	ID/EX.A←Regs[IF/ID.IR[rs1]]; ID/EX.B←Regs[IF/ID.IR[rs2]]; ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR; ID/EX.Imm←sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load instruction**	**Branch instruction**
EX	EX/MEM.IR←ID/EX.IR; EX/MEM.ALUOutput← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput← ID/EX.A+ID/EX.Imm; EX/MEM.B←ID/EX.B;	EX/MEM.ALUOutput← ID/EX.NPC + (ID/EX.Imm<< 2); EX/MEM.cond← (ID/EX.A == ID/EX.B);
MEM	MEM/WB.IR←EX/MEM.IR; MEM/WB.ALUOutput← EX/MEM.ALUOutput;	MEM/WB.IR←EX/MEM.IR; MEM/WB.LMD← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput]← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]]← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]]← MEM/WB.LMD;	

Figure C.20 Events on every pipe stage of the RISC V pipeline. Let's review the actions in the stages that are specific to the pipeline organization. In IF, in addition to fetching the instruction and computing the new PC, we store the incremented PC both into the PC and into a pipeline register (NPC) for later use in computing the branch-target address. This structure is the same as the organization in Figure C.19, where the PC is updated in IF from one of two sources. In ID, we fetch the registers, extend the sign of the 12 bits of the IR (the immediate field), and pass along the IR and NPC. During EX, we perform an ALU operation or an address calculation; we pass along the IR and the B register (if the instruction is a store). We also set the value of cond to 1 if the instruction is a taken branch. During the MEM phase, we cycle the memory, write the PC if needed, and pass along values needed in the final pipe stage. Finally, during WB, we update the register field from either the ALU output or the loaded value. For simplicity we always pass the entire IR from one stage to the next, although as an instruction proceeds down the pipeline, less and less of the IR is needed.

of data from one stage to the next. Notice that the actions in the first two stages are independent of the current instruction type; they must be independent because the instruction is not decoded until the end of the ID stage. The IF activity depends on whether the instruction in EX/MEM is a taken branch. If so, then the branch-target address of the branch instruction in EX/MEM is written into the PC at the end of IF; otherwise, the incremented PC will be written back. (As we said earlier, this effect of branches leads to complications in the pipeline that we deal with in the next few sections.) The fixed-position encoding of the register source operands is critical to allowing the registers to be fetched during ID.

To control this simple pipeline we need only determine how to set the control for the four multiplexers in the data path of Figure C.19. The two multiplexers in the ALU stage are set depending on the instruction type, which is dictated by the IR field of the ID/EX register. The top ALU input multiplexer is set by whether the instruction is a branch or not, and the bottom multiplexer is set by whether the instruction is a register-register ALU operation or any other type of operation. The multiplexer in the IF stage chooses whether to use the value of the incremented PC or the value of the EX/MEM.ALUOutput (the branch target) to write into the PC. This multiplexer is controlled by the field EX/MEM.cond. The fourth multiplexer is controlled by whether the instruction in the WB stage is a load or an ALU operation. In addition to these four multiplexers, there is one additional multiplexer needed that is not drawn in Figure C.19, but whose existence is clear from looking at the WB stage of an ALU operation. The destination register field is in one of two different places depending on the instruction type (register-register ALU versus either ALU immediate or load). Thus, we will need a multiplexer to choose the correct portion of the IR in the MEM/WB register to specify the register destination field, assuming the instruction writes a register.

Implementing the Control for the RISC V Pipeline

The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called *instruction issue*; an instruction that has made this step is said to have *issued*. For the RISC V integer pipeline, all the data hazards can be checked during the ID phase of the pipeline. If a data hazard exists, the instruction is stalled before it is issued. Likewise, we can determine what forwarding will be needed during ID and set the appropriate controls then. Detecting interlocks early in the pipeline reduces the hardware complexity because the hardware never has to suspend an instruction that has updated the state of the processor, unless the entire processor is stalled. Alternatively, we can detect the hazard or forwarding at the beginning of a clock cycle that uses an operand (EX and MEM for this pipeline). To show the differences in these two approaches, we will show how the interlock for a read after write (RAW) hazard with the source coming from a load instruction (called a *load interlock*) can be implemented by a check in ID, while the implementation of forwarding paths to

Situation	Example code sequence	Action

No dependence	`ld   x1,45(x2)`
	`add  x5,x6,x7`
	`sub  x8,x6,x7`
	`or   x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

Dependence requiring stall	`ld   x1,45(x2)`
	`add  x5,x1,x7`
	`sub  x8,x6,x7`
	`or   x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld   x1,45(x2)`
	`add  x5,x6,x7`
	`sub  x8,x1,x7`
	`or   x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld   x1,45(x2)`
	`add  x5,x6,x7`
	`sub  x8,x6,x7`
	`or   x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

Figure C.21 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case once execution continues (dependence overcome by forwarding). Of course, hazards that involve x0 can be ignored because the register always contains 0, and the preceding test could be extended to do this.

the ALU inputs can be done during EX. Figure C.21 lists the variety of circumstances that we must handle.

Let's start with implementing the load interlock. If there is a RAW hazard with the source instruction being a load, the load instruction will be in the EX stage when an instruction that needs the load data will be in the ID stage. Thus, we can describe all the possible hazard situations with a small table, which can be directly translated to an implementation. Figure C.22 shows a table that detects all load interlocks when the instruction using the load result is in the ID stage.

Once a hazard has been detected, the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages from advancing. As we said earlier, all the control information is carried in the pipeline registers. (Carrying the instruction along is enough, because all control is derived from it.) Thus, when we detect a hazard we need only change the control portion of the ID/EX pipeline register to all 0s, which happens to be a no-op (an instruction that does nothing, such as `add x0,x0,x0`). In addition, we simply recirculate the contents of the IF/ID registers to hold the stalled instruction. In a pipeline with more complex hazards, the same ideas would apply: we can detect the hazard by comparing some set of pipeline registers and shift in no-ops to prevent erroneous execution.

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..6}$)	Matching operand fields
Load	Register-register ALU, load, store, ALU immediate, or branch	`ID/EX.IR[rd]` == `IF/ID.IR[rs1]`
Load	Register-register ALU, or branch	`ID/EX.IR[rd]` == `IF/ID.IR[rs2]`

Figure C.22 The logic to detect the need for load interlocks during the ID stage of an instruction requires two comparisons, one for each possible source. Remember that the IF/ID register holds the state of the instruction in ID, which potentially uses the load result, while ID/EX holds the state of the instruction in EX, which is the load instruction.

Implementing the forwarding logic is similar, although there are more cases to consider. The key observation needed to implement the forwarding logic is that the pipeline registers contain both the data to be forwarded as well as the source and destination register fields. All forwarding logically happens from the ALU or data memory output to the ALU input, the data memory input, or the zero detection unit. Thus, we can implement the forwarding by a comparison of the destination registers of the IR contained in the EX/MEM and MEM/WB stages against the source registers of the IR contained in the ID/EX and EX/MEM registers. Figure C.23 shows the comparisons and possible forwarding operations where the destination of the forwarded result is an ALU input for the instruction currently in EX.

In addition to the comparators and combinational logic that we must determine when a forwarding path needs to be enabled, we also must enlarge the multiplexers at the ALU inputs and add the connections from the pipeline registers that are used to forward the results. Figure C.24 shows the relevant segments of the pipelined data path with the additional multiplexers and connections in place.

For RISC V, the hazard detection and forwarding hardware is reasonably simple; we will see that things become somewhat more complicated when we extend this pipeline to deal with floating point. Before we do that, we need to handle branches.

Dealing With Branches in the Pipeline

In RISC V, conditional branches depend on comparing two register values, which we assume occurs during the EX cycle, and uses the ALU for this function. We will need to also compute the branch target address. Because testing the branch condition and determining the next PC will determine what the branch penalty is, we would like to compute both the possible PCs and choose the correct PC before the end of the EX cycle. We can do this by adding a separate adder that computes the branch target address during ID. Because the instruction is not yet decoded, we will be computing a possible target as if every instruction were a branch. This is

Pipeline register of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs1]`
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs2]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs1]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs2]`

Figure C.23 Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB. There are 10 separate comparisons needed to tell whether a forwarding operation should occur. The top and bottom ALU inputs refer to the inputs corresponding to the first and second ALU source operands, respectively, and are shown explicitly in Figure C.18 on page C.30 and in Figure C.24 on page C.36. Remember that the pipeline latch for destination instruction in EX is ID/EX, while the source values come from the ALUOutput portion of EX/MEM or MEM/WB or the LMD portion of MEM/WB. There is one complication not addressed by this logic: dealing with multiple instructions that write the same register. For example, during the code sequence `add x1, x2, x3; addi x1, x1, 2; sub x4, x3, x1`, the logic must ensure that the `sub` instruction uses the result of the `addi` instruction rather than the result of the `add` instruction. The logic shown here can be extended to handle this case by simply testing that forwarding from MEM/WB is enabled only when forwarding from EX/MEM is not enabled for the same input. Because the `addi` result will be in EX/MEM, it will be forwarded, rather than the `add` result in MEM/WB.

likely faster than computing the target and evaluating the condition both in EX, but does use slightly more energy.

Figure C.25 shows a pipelined data path assuming the adder in ID and the evaluation of the branch condition in EX, a minor change of the pipeline structure. This pipeline will incur a two-cycle penalty on branches. In some early RISC processors, such as MIPS, the condition test on branches was restricted to allow the test to occur in ID, reducing the branch delay to one cycle. Of course, that meant that an ALU operation to a register followed by a conditional branch based on that register incurred a data hazard, which does not occur if the branch condition is evaluated in EX.

As pipeline depths increased, the branch delay increased, which made dynamic branch prediction necessary. For example, a processor with separate decode and

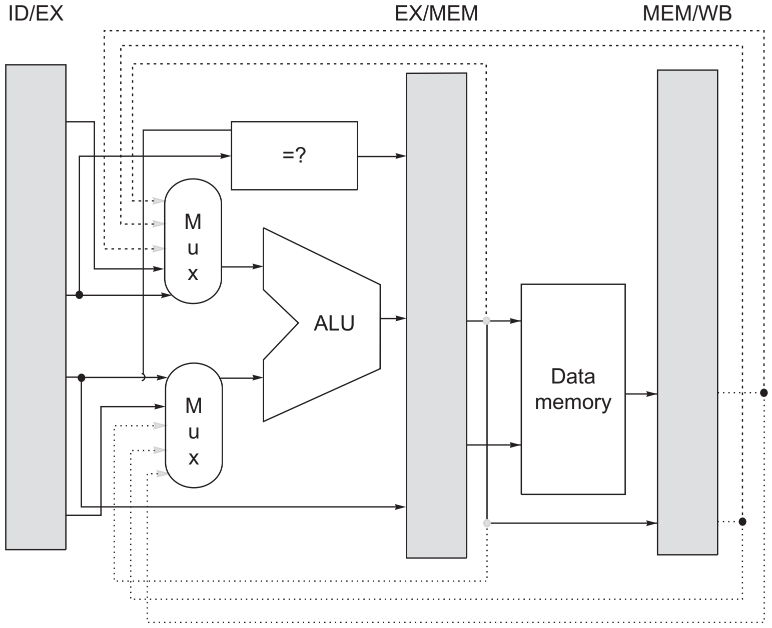


Figure C.24 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

register fetch stages will probably have a branch delay that is at least 1 clock cycle longer. The branch delay, unless it is dealt with, turns into a branch penalty. Many older processors that implement more complex instruction sets have branch delays of 4 clock cycles or more, and large, deeply pipelined processors often have branch penalties of 6 or 7. Aggressive high-end superscalars, such as the Intel i7 discussed in Chapter 3, may have branch penalties of 10–15 cycles! In general, the deeper the pipeline, the worse the branch penalty in clock cycles, and the more critical that branches be accurately predicted.

C.4 What Makes Pipelining Hard to Implement?

Now that we understand how to detect and resolve hazards, we can deal with some complications that we have avoided so far. The first part of this section considers the challenges of exceptional situations where the instruction execution order is changed in unexpected ways. In the second part of this section, we discuss some of the challenges raised by different instruction sets.

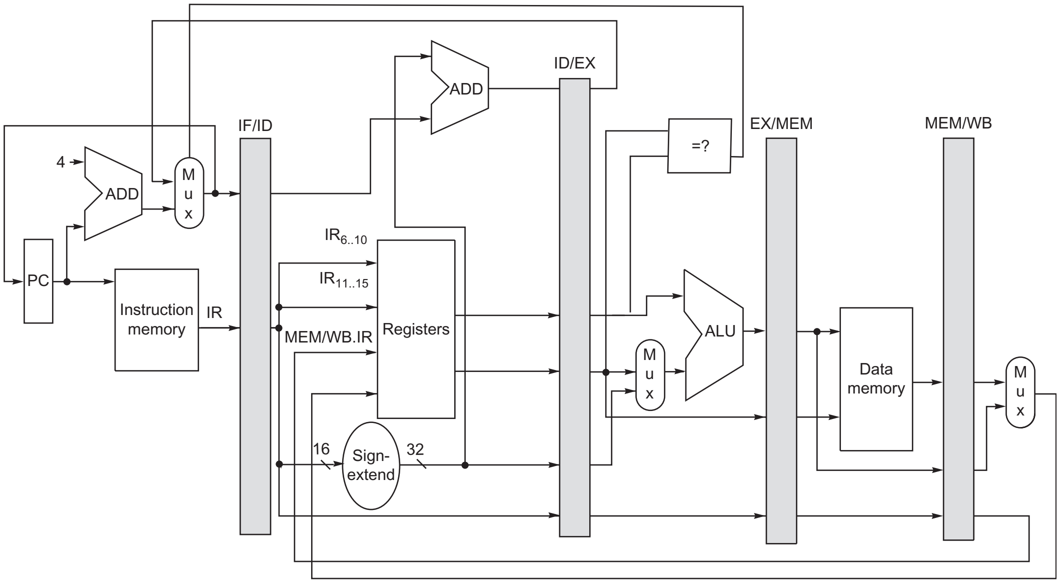


Figure C.25 To minimize the impact of deciding whether a conditional branch is taken, we compute the branch target address in ID while doing the conditional test and final selection of next PC in EX. As mentioned in Figure C.19, the PC can be thought of as a pipeline register (e.g., as part of ID/IF), which is written with the address of the next instruction at the end of each IF cycle.

Dealing With Exceptions

Exceptional situations are harder to handle in a pipelined processor because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the processor. In a pipelined processor, an instruction is executed piece by piece and is not completed for several clock cycles. Unfortunately, other instructions in the pipeline can raise exceptions that may force the processor to abort the instructions in the pipeline before they complete. Before we discuss these problems and their solutions in detail, we need to understand what types of situations can arise and what architectural requirements exist for supporting them.

Types of Exceptions and Requirements

The terminology used to describe exceptional situations where the normal execution order of instruction is changed varies among processors. The terms *interrupt, fault*, and *exception* are used, although not in a consistent fashion. We use the term *exception* to cover all these mechanisms, including the following:

- I/O device request
- Invoking an operating system service from a user program

- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

When we wish to refer to some particular class of such exceptions, we will use a longer name, such as I/O interrupt, floating-point exception, or page fault.

Although we use the term *exception* to cover all of these events, individual events have important characteristics that determine what action is needed in the hardware. The requirements on exceptions can be characterized on five semi-independent axes:

1. *Synchronous versus asynchronous*—If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is *synchronous*. With the exception of hardware malfunctions, *asynchronous* events are caused by devices external to the processor and memory. Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.
2. *User requested versus coerced*—If the user task directly asks for it, it is a *user-requested* event. In some sense, user-requested exceptions are not really exceptions, because they are predictable. They are treated as exceptions, however, because the same mechanisms that are used to save and restore the state are used for these user-requested events. Because the only function of an instruction that triggers this exception is to cause the exception, user-requested exceptions can always be handled after the instruction has completed. *Coerced* exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable.
3. *User maskable versus user nonmaskable*—If an event can be masked or disabled by a user task, it is *user maskable*. This mask simply controls whether the hardware responds to the exception or not.
4. *Within versus between instructions*—This classification depends on whether the event prevents instruction completion by occurring in the middle of execution—no matter how short—or whether it is recognized *between* instructions. Exceptions that occur *within* instructions are usually synchronous, because the instruction triggers the exception. It's harder to implement exceptions that

occur within instructions than those between instructions, because the instruction must be stopped and restarted. Asynchronous exceptions that occur within instructions arise from catastrophic situations (e.g., hardware malfunction) and always cause program termination.

5. *Resume versus terminate*—If the program's execution always stops after the interrupt, it is a *terminating* event. If the program's execution continues after the interrupt, it is a *resuming* event. It is easier to implement exceptions that terminate execution, because the processor need not be able to restart execution of the same program after handling the exception.

Figure C.26 classifies the preceding examples according to these five categories. The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed. Implementing such exceptions requires that another program must be invoked to save the state of the executing program, correct the cause of the exception, and then restore the state of the program before the

Exception type	**Synchronous vs. asynchronous**	**User request vs. coerced**	**User maskable vs. nonmaskable**	**Within vs. between instructions**	**Resume vs. terminate**
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure C.26 Five categories are used to define what actions are needed for the different exception types. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, processors should be able to resume after such exceptions.

instruction that caused the exception can be tried again. This process must be effectively invisible to the executing program. If a pipeline provides the ability for the processor to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or processor is said to be *restartable*. While early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it is needed to implement virtual memory (see Chapter 2).

Stopping and Restarting Execution

As in unpipelined implementations, the most difficult exceptions have two properties: (1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages), and (2) they must be restartable. In our RISC V pipeline, for example, a virtual memory page fault resulting from a data fetch cannot occur until sometime in the MEM stage of the instruction. By the time that fault is seen, several other instructions will be in execution. A page fault must be restartable and requires the intervention of another process, such as the operating system. Thus, the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state. Restarting is usually implemented by saving the PC of the instruction at which to restart. If the restarted instruction is not a branch, then we will continue to fetch the sequential successors and begin their execution in the normal fashion. If the restarted instruction is a branch, then we will reevaluate the branch condition and begin fetching from either the target or the fall-through. When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next IF.
2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline; this can be done by placing zeros into the pipeline latches of all instructions in the pipeline, starting with the instruction that generates the exception, but not those that precede that instruction. This prevents any state changes for instructions that will not be completed before the exception is handled.
3. After the exception-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

After the exception has been handled, special instructions return the processor from the exception by reloading the PCs and restarting the instruction stream (using the exception return in RISC V). If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have *precise exceptions*. Ideally, the faulting instruction would not have changed the state, and correctly handling some exceptions requires that the faulting instruction have no effects. For other exceptions, such as floating-point exceptions, the faulting instruction on some processors writes its result before the exception can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is identical to one of the source operands. Because floating-point operations may run for many cycles, it is highly likely that some other instruction may have written the source operands (as we will see in the next section, floating-point operations often complete out of order). To overcome this, many recent high-performance processors have introduced two modes of operation. One mode has precise exceptions and the other (fast or performance mode) does not. Of course, the precise exception mode is slower, since it allows less overlap among floating-point instructions.

Supporting precise exceptions is a requirement in many systems, while in others it is "just" valuable because it simplifies the operating system interface. At a minimum, any processor with demand paging or IEEE arithmetic trap handlers must make its exceptions precise, either in the hardware or with some software support. For integer pipelines, the task of creating precise exceptions is easier, and accommodating virtual memory strongly motivates the support of precise exceptions for memory references. In practice, these reasons have led designers and architects to always provide precise exceptions for the integer pipeline. In this section we describe how to implement precise exceptions for the RISC V integer pipeline. We will describe techniques for handling the more complex challenges arising in the floating-point pipeline in Section C.5.

Exceptions in RISC V

Figure C.27 shows the RISC V pipeline stages and which problem exceptions might occur in each stage. With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution. For example, consider this instruction sequence:

`ld`	IF	ID	EX	MEM	WB	
`add`		IF	ID	EX	MEM	WB

Pipeline stage	**Problem exceptions occurring**
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure C.27 Exceptions that may occur in the RISC V pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

This pair of instructions can cause a data page fault and an arithmetic exception at the same time, because the `ld` is in the MEM stage while the `add` is in the EX stage. This case can be handled by dealing with only the data page fault and then restarting the execution. The second exception will reoccur (but not the first, if the software is correct), and when the second exception occurs it can be handled independently.

In reality, the situation is not as straightforward as this simple example. Exceptions may occur out of order; that is, an instruction may cause an exception before an earlier instruction causes one. Consider again the preceding sequence of instructions, `ld` followed by `add`. The `ld` can get a data page fault, seen when the instruction is in MEM, and the `add` can get an instruction page fault, seen when the `add` instruction is in IF. The instruction page fault will actually occur first, even though it is caused by a later instruction!

Because we are implementing precise exceptions, the pipeline is required to handle the exception caused by the `ld` instruction first. To explain how this works, let's call the instruction in the position of the `ld` instruction i, and the instruction in the position of the `add` instruction $i+1$. The pipeline cannot simply handle an exception when it occurs in time, because that will lead to exceptions occurring out of the unpipelined order. Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction. The exception status vector is carried along as the instruction goes down the pipeline. Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off (this includes both register writes and memory writes). Because a store can cause an exception during MEM, the hardware must be prepared to prevent the store from completing if it raises an exception.

When an instruction enters WB (or is about to leave MEM), the exception status vector is checked. If any exceptions are posted, they are handled in the order in which they would occur in time on an unpipelined processor—the exception corresponding to the earliest instruction (and usually the earliest pipe stage for that instruction) is handled first. This guarantees that all exceptions will be seen on instruction i before any are seen on $i+1$. Of course, any action taken in earlier pipe stages on behalf of instruction i may be invalid, but because writes to the register file and memory were disabled, no state could have been changed. As we will see in Section C.5, maintaining this precise model for FP operations is much harder.

In the next subsection we describe problems that arise in implementing exceptions in the pipelines of processors with more powerful, longer-running instructions.

Instruction Set Complications

No RISC V instruction has more than one result, and our RISC V pipeline writes that result only at the end of an instruction's execution. When an instruction is guaranteed to complete, it is called *committed*. In the RISC V integer pipeline, all instructions are committed when they reach the end of the MEM stage (or beginning of WB) and no instruction updates the state before that stage. Thus, precise exceptions are straightforward. Some processors have instructions that change the state in the middle of the instruction execution, before the instruction and its predecessors are guaranteed to complete. For example, autoincrement addressing modes in the IA-32 architecture cause the update of registers in the middle of an instruction execution. In such a case, if the instruction is aborted because of an exception, it will leave the processor state altered. Although we know which instruction caused the exception, without additional hardware support the exception will be imprecise because the instruction will be half finished. Restarting the instruction stream after such an imprecise exception is difficult. Alternatively, we could avoid updating the state before the instruction commits, but this may be difficult or costly, because there may be dependences on the updated state: consider a VAX instruction that autoincrements the same register multiple times. Thus, to maintain a precise exception model, most processors with such instructions have the ability to back out any state changes made before the instruction is committed. If an exception occurs, the processor uses this ability to reset the state of the processor to its value before the interrupted instruction started. In the next section, we will see that a more powerful RISC V floating-point pipeline can introduce similar problems, and Section C.7 introduces techniques that substantially complicate exception handling.

A related source of difficulties arises from instructions that update memory state during execution, such as the string copy operations on the Intel architecture or IBM 360 (see Appendix K). To make it possible to interrupt and restart these instructions, the instructions are defined to use the general-purpose registers as working registers. Thus, the state of the partially completed instruction is always in the registers, which are saved on an exception and restored after the exception, allowing the instruction to continue.

A different set of difficulties arises from odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore. Condition codes are a good example of this. Many processors set the condition codes implicitly as part of the instruction. This approach has advantages, because condition codes decouple the evaluation of the condition from the actual branch. However, implicitly set condition codes can cause difficulties in scheduling any pipeline delays between setting the condition code and the branch, because most instructions set the condition code and cannot be used in the delay slots between the condition evaluation and the branch.

Additionally, in processors with condition codes, the processor must decide when the branch condition is fixed. This involves finding out when the condition code has been set for the last time before the branch. In most processors with implicitly set condition codes, this is done by delaying the branch condition evaluation until all previous instructions have had a chance to set the condition code.

Of course, architectures with explicitly set condition codes allow the delay between condition test and the branch to be scheduled; however, pipeline control must still track the last instruction that sets the condition code to know when the branch condition is decided. In effect, the condition code must be treated as an operand that requires hazard detection for RAW hazards with branches, just as RISC V must do on the registers.

A final thorny area in pipelining is multicycle operations. Imagine trying to pipeline a sequence of x86 instructions such as this:

```
mov       BX, AX         ; moves between registers
add       42(BX+SI),BX;adds memory contents and register
                         ;  to same memory location
sub       BX,AX          ;subtracts registers
rep movsb                ;moves a character string of
                         ; length given by register CX
```

Although none of these instructions is particularly long (an x86 instruction can be up to 15 bytes), they do differ radically in the number of clock cycles they will require, from as low as one up to hundreds of clock cycles. These instructions also require different numbers of data memory accesses, from zero to possibly hundreds. The data hazards are very complex and occur both between and within instructions (nothing prevents the movsb from having an overlapping source and destination!). The simple solution of making all instructions execute for the same number of clock cycles is unacceptable because it introduces an enormous number of hazards and bypass conditions and makes an immensely long pipeline. Pipelining the x86 at the instruction level is difficult, but a clever solution was found, similar to one used for the VAX. They pipeline the *microinstruction* execution; a microinstruction is a simple instruction used in sequences to implement a more complex instruction set. Because the microinstructions are simple (they look a lot like RISC V), the pipeline control is much easier. Since 1995, all Intel IA-32 microprocessors have used this strategy of converting the IA-32 instructions into microoperations, and then pipelining the microoperations. In fact, this approach is even used for some of the more complex instructions in the ARM architecture.

In comparison, load-store processors have simple operations with similar amounts of work and pipeline more easily. If architects realize the relationship between instruction set design and pipelining, they can design architectures for more efficient pipelining. In the next section, we will see how the RISC V pipeline deals with long-running instructions, specifically floating-point operations.

For many years, the interaction between instruction sets and implementations was believed to be small, and implementation issues were not a major focus in designing instruction sets. In the 1980s, it became clear that the difficulty and inefficiency of pipelining could both be increased by instruction set complications. In the 1990s, all companies moved to simpler instructions sets with the goal of reducing the complexity of aggressive implementations.

C.5 Extending the RISC V Integer Pipeline to Handle Multicycle Operations

We now want to explore how our RISC V pipeline can be extended to handle floating-point operations. This section concentrates on the basic approach and the design alternatives, closing with some performance measurements of a RISC V floating-point pipeline.

It is impractical to require that all RISC V FP operations complete in 1 clock cycle, or even in 2. Doing so would mean accepting a slow clock or using enormous amounts of logic in the FP units, or both. Instead, the FP pipeline will allow for a longer latency for operations. This is easier to grasp if we imagine the FP instructions as having the same pipeline as the integer instructions, with two important changes. First, the EX cycle may be repeated as many times as needed to complete the operation—the number of repetitions can vary for different operations. Second, there may be multiple FP functional units. A stall will occur if the instruction to be issued will cause either a structural hazard for the functional unit it uses or a data hazard.

For this section, let's assume that there are four separate functional units in our RISC V implementation:

1. The main integer unit that handles loads and stores, integer ALU operations, and branches
2. FP and integer multiplier
3. FP adder that handles FP add, subtract, and conversion
4. FP and integer divider

If we also assume that the execution stages of these functional units are not pipelined, then Figure C.28 shows the resulting pipeline structure. Because EX is not pipelined, no other instruction using that functional unit may issue until the previous instruction leaves EX. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.

In reality, the intermediate results are probably not cycled around the EX unit as Figure C.28 suggests; instead, the EX pipeline stage has some number of clock delays larger than 1. We can generalize the structure of the FP pipeline shown in Figure C.28 to allow pipelining of some stages and multiple ongoing operations. To describe such a pipeline, we must define both the latency of the functional units and also the *initiation interval* or *repeat interval*. We define latency the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type. For example, we will use the latencies and initiation intervals shown in Figure C.29.

With this definition of latency, integer ALU operations have a latency of 0, because the results can be used on the next clock cycle, and loads have a latency of 1, because their results can be used after one intervening cycle. Because most operations consume their operands at the beginning of EX, the latency is usually the number of stages after EX that an instruction produces a result—for example, zero stages for ALU operations and one stage for loads. The primary exception is stores, which consume the value being stored one cycle later. Hence, the latency to

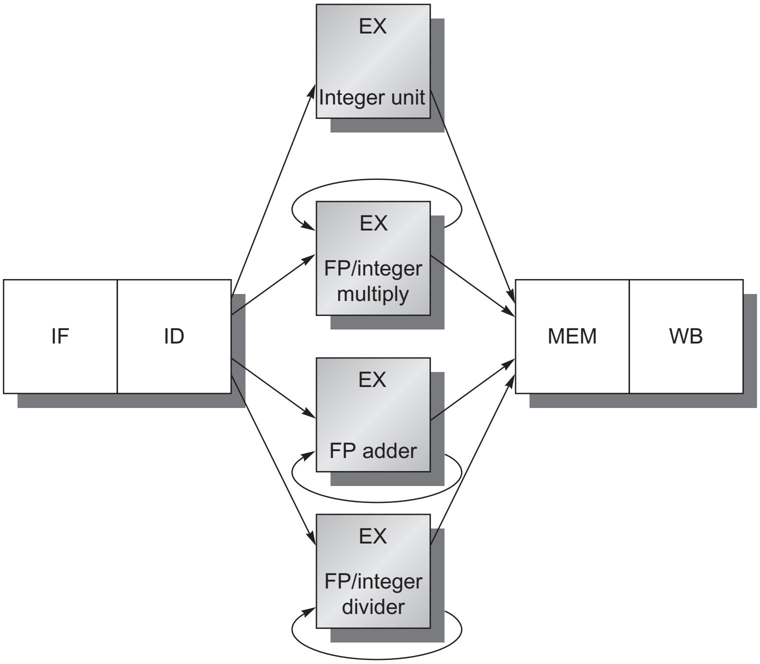


Figure C.28 The RISC V pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.29 Latencies and initiation intervals for functional units.

a store for the value being stored, but not for the base address register, will be one cycle less. Pipeline latency is essentially equal to one cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result. Thus, for the preceding example pipeline, the number of stages in an FP add is four, while the number of stages in an FP multiply is seven. To achieve a higher clock rate, designers need to put fewer logic levels in each pipe stage, which makes the number of pipe stages required for more complex operations larger. The penalty for the faster clock rate is thus longer latency for operations.

The example pipeline structure in Figure C.29 allows up to four outstanding FP adds, seven outstanding FP/integer multiplies, and one FP divide. Figure C.30 shows how this pipeline can be drawn by extending Figure C.28. The repeat interval is implemented in Figure C.30 by adding additional pipeline stages, which will be separated by additional pipeline registers. Because the units are independent, we name the stages differently. The pipeline stages that take multiple clock cycles, such as the divide unit, are further subdivided to show the latency of those stages. Because they are not complete stages, only one operation may be active. The pipeline structure can also be shown using the familiar diagrams from earlier in the appendix, as Figure C.31 shows for a set of independent FP operations and FP loads and stores. Naturally, the longer latency of the FP operations increases the frequency of RAW hazards and resultant stalls, as we will see later in this section.

The structure of the pipeline in Figure C.30 requires the introduction of the additional pipeline registers (e.g.,A1/A2, A2/A3, A3/A4) and the modification of the connections to those registers. The ID/EX register must be expanded to

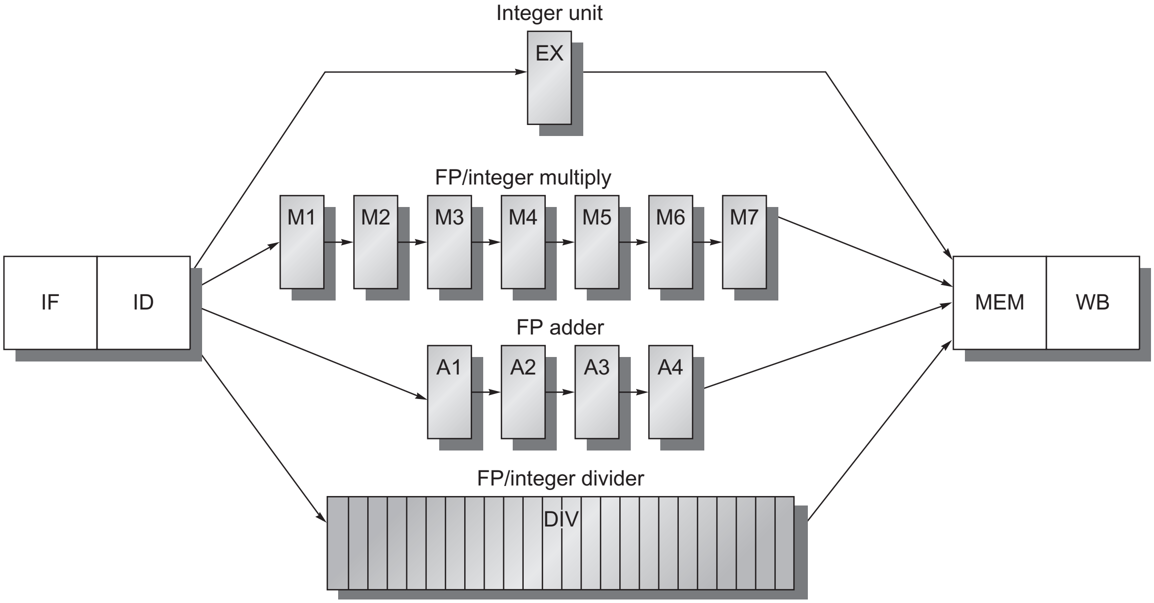


Figure C.30 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

`fmul.d`	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
`fadd.d`		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
`fadd.d`			IF	ID	*EX*	**MEM**	WB				
`fsd`				IF	ID	*EX*	**MEM**	WB			

Figure C.31 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

connect ID to EX, DIV, M1, and A1; we can refer to the portion of the register associated with one of the next stages with the notation ID/EX, ID/DIV, ID/M1, or ID/A1. The pipeline register between ID and all the other stages may be thought of as logically separate registers and may, in fact, be implemented as separate registers. Because only one operation can be in a pipe stage at a time, the control information can be associated with the register at the head of the stage.

Hazards and Forwarding in Longer Latency Pipelines

There are a number of different aspects to the hazard detection and forwarding for a pipeline like that shown in Figure C.30.

1. Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.
2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.
3. Write after write (WAW) hazards are possible, because instructions no longer reach WB in order. Note that write after read (WAR) hazards are not possible, because the register reads always occur in ID.
4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.
5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

The increase in stalls arising from longer operation latencies is fundamentally the same as that for the integer pipeline. Before describing the new problems that arise in this FP pipeline and looking at solutions, let's examine the potential impact of RAW hazards. Figure C.32 shows a typical FP code sequence and the resultant stalls. At the end of this section, we'll examine the performance of this FP pipeline for our SPEC subset.

Now look at the problems arising from writes, described as (2) and (3) in the earlier list. If we assume that the FP register file has one write port, sequences of FP operations, as well as an FP load together with FP operations, can cause conflicts

	Clock cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Figure C.32 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The `fsd` must be stalled an extra cycle so that its MEM does not conflict with the `fadd.d`. Extra hardware could easily handle this case.

for the register write port. Consider the pipeline sequence shown in Figure C.33. In clock cycle 11, all three instructions will reach WB and want to write the register file. With only a single register file write port, the processor must serialize the instruction completion. This single register port represents a structural hazard. We could increase the number of write ports to solve this, but that solution may be unattractive because the additional write ports would be used only rarely. This is because the maximum steady-state number of write ports needed is 1. Instead, we choose to detect and enforce access to the write port as a structural hazard.

There are two different ways to implement this interlock. The first is to track the use of the write port in the ID stage and to stall an instruction before it issues, just as we would for any other structural hazard. Tracking the use of the write port can be done with a shift register that indicates when already-issued instructions will use the register file. If the instruction in ID needs to use the register file at the same time

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the `fmul.d`, `fadd.d`, and `fld` are in the MEM stage in clock cycle 10, only the `fld` actually uses the memory, so no structural hazard exists for MEM.

as an instruction already issued, the instruction in ID is stalled for a cycle. On each clock the reservation register is shifted 1 bit. This implementation has an advantage: It maintains the property that all interlock detection and stall insertion occurs in the ID stage. The cost is the addition of the shift register and write conflict logic. We will assume this scheme throughout this section.

An alternative scheme is to stall a conflicting instruction when it tries to enter either the MEM or WB stage. If we wait to stall the conflicting instructions until they want to enter the MEM or WB stage, we can choose to stall either instruction. A simple, though sometimes suboptimal, heuristic is to give priority to the unit with the longest latency, because that is the one most likely to have caused another instruction to be stalled for a RAW hazard. The advantage of this scheme is that it does not require us to detect the conflict until the entrance of the MEM or WB stage, where it is easy to see. The disadvantage is that it complicates pipeline control, as stalls can now arise from two places. Notice that stalling before entering MEM will cause the EX, A4, or M7 stage to be occupied, possibly forcing the stall to trickle back in the pipeline. Likewise, stalling before WB would cause MEM to back up.

Our other problem is the possibility of WAW hazards. To see that these exist, consider the example in Figure C.33. If the `fadd.d` instruction were issued one cycle earlier and had a destination of `f2`, then it would create a WAW hazard, because it would write `f2` one cycle earlier than the `fadd.d`. Note that this hazard only occurs when the result of the `fadd.d` is overwritten *without* any instruction ever using it! If there were a use of f2 between the `fadd.d` and the `fadd.d`, the pipeline would need to be stalled for a RAW hazard, and the `fadd.d` would not issue until the `fadd.d` was completed. We could argue that, for our pipeline, WAW hazards only occur when a useless instruction is executed, but we must still detect them and make sure that the result of the `fadd.d` appears in `f2` when we are done. (As we will see in Section C.8, such sequences sometimes *do* occur in reasonable code.)

There are two possible ways to handle this WAW hazard. The first approach is to delay the issue of the load instruction until the `fadd.d` enters MEM. The second approach is to stamp out the result of the `fadd.d` by detecting the hazard and changing the control so that the `fadd.d` does not write its result. Then the `fadd.d` can issue right away. Because this hazard is rare, either scheme will work fine—you can pick whatever is simpler to implement. In either case, the hazard can be detected during ID when the `fadd.d` is issuing, and stalling the `fadd.d` or making the `fadd.d` a no-op is easy. The difficult situation is to detect that the `fadd.d` might finish before the `fadd.d`, because that requires knowing the length of the pipeline and the current position of the `fadd.d`. Luckily, this code sequence (two writes with no intervening read) will be very rare, so we can use a simple solution: if an instruction in ID wants to write the same register as an instruction already issued, do not issue the instruction to EX. In Section C.7, we will see how additional hardware can eliminate stalls for such hazards. First, let's put together the pieces for implementing the hazard and issue logic in our FP pipeline.

In detecting the possible hazards, we must consider hazards among FP instructions, as well as hazards between an FP instruction and an integer instruction. Except for FP loads-stores and FP-integer register moves, the FP and integer registers are distinct. All integer instructions operate on the integer registers, while the FP operations operate only on their own registers. Thus, we need only consider FP loads-stores and FP register moves in detecting hazards between FP and integer instructions. This simplification of pipeline control is an additional advantage of having separate register files for integer and floating-point data. (The main advantages are a doubling of the number of registers, without making either set larger, and an increase in bandwidth without adding more ports to either set. The main disadvantage, beyond the need for an extra register file, is the small cost of occasional moves needed between the two register sets.) Assuming that the pipeline does all hazard detection in ID, there are three checks that must be performed before an instruction can issue:

1. *Check for structural hazards*—Wait until the required functional unit is not busy (this is only needed for divides in this pipeline) and make sure the register write port is available when it will be needed.
2. *Check for a RAW data hazard*—Wait until the source registers are not listed as pending destinations in a pipeline register that will not be available when this instruction needs the result. A number of checks must be made here, depending on both the source instruction, which determines when the result will be available, and the destination instruction, which determines when the value is needed. For example, if the instruction in ID is an FP operation with source register f2, then f2 cannot be listed as a destination in ID/A1, A1/A2, or A2/A3, which correspond to FP add instructions that will not be finished when the instruction in ID needs a result. (ID/A1 is the portion of the output register of ID that is sent to A1.) Divide is somewhat more tricky, if we want to allow the last few cycles of a divide to be overlapped, because we need to handle the case when a divide is close to finishing as special. In practice, designers might ignore this optimization in favor of a simpler issue test.
3. *Check for a WAW data hazard*—Determine if any instruction in A1, … , A4, D, M1, … , M7 has the same register destination as this instruction. If so, stall the issue of the instruction in ID.

Although the hazard detection is more complex with the multicycle FP operations, the concepts are the same as for the RISC V integer pipeline. The same is true for the forwarding logic. The forwarding can be implemented by checking if the destination register in any of the EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB registers is one of the source registers of a floating-point instruction. If so, the appropriate input multiplexer will have to be enabled so as to choose the forwarded data. In the exercises, you will have the opportunity to specify the logic for the RAW and WAW hazard detection as well as for forwarding.

Multicycle FP operations also introduce problems for our exception mechanisms, which we deal with next.

Maintaining Precise Exceptions

Another problem caused by these long-running instructions can be illustrated with the following sequence of code:

```
fdiv.d     f0,f2,f4
fadd.d     f10,f10,f8
fsub.d     f12,f12,f14
```

This code sequence looks straightforward; there are no dependences. A problem arises, however, because an instruction issued early may complete after an instruction issued later. In this example, we can expect `fadd.d` and `fsub.d` to complete *before* the `fdiv.d` completes. This is called *out-of-order completion* and is common in pipelines with long-running operations (see Section C.7). Because hazard detection will prevent any dependence among instructions from being violated, why is out-of-order completion a problem? Suppose that the `fsub.d` causes a floating-point arithmetic exception at a point where the `fadd.d` has completed but the `fdiv.d` has not. The result will be an imprecise exception, something we are trying to avoid. It may appear that this could be handled by letting the floating-point pipeline drain, as we do for the integer pipeline. But the exception may be in a position where this is not possible. For example, if the fdiv.d decided to take a floating-point-arithmetic exception after the add completed, we could not have a precise exception at the hardware level. In fact, because the fadd.d destroys one of its operands, we could not restore the state to what it was before the fdiv.d, even with software help.

This problem arises because instructions are completing in a different order than they were issued. There are four possible approaches to dealing with out-of-order completion. The first is to ignore the problem and settle for imprecise exceptions. This approach was used in the 1960s and early 1970s. It was still used in some supercomputers in thepast fifteen years, where certain classes of exceptions were not allowed or were handled by the hardware without stopping the pipeline. It is difficult to use this approach in most processors built today because of features such as virtual memory and the IEEE floating-point standard that essentially require precise exceptions through a combination of hardware and software. As mentioned earlier, some recent processors have solved this problem by introducing two modes of execution: a fast, but possibly imprecise mode and a slower, precise mode. The slower precise mode is implemented either with a mode switch or by insertion of explicit instructions that test for FP exceptions. In either case, the amount of overlap and reordering permitted in the FP pipeline is significantly restricted so that effectively only one FP instruction is active at a time. This solution was used in the DEC Alpha 21064 and 21164, in the IBM Power1 and Power2, and in the MIPS R8000.

A second approach is to buffer the results of an operation until all the operations that were issued earlier are complete. Some processors actually use this solution, but it becomes expensive when the difference in running times among operations is large, because the number of results to buffer can become large. Furthermore, results from the queue must be bypassed to continue issuing instructions while waiting for the longer instruction. This requires a large number of comparators and a very large multiplexer.

There are two viable variations on this basic approach. The first is a *history file*, used in the CYBER 180/990. The history file keeps track of the original values of registers. When an exception occurs and the state must be rolled back earlier than some instruction that completed out of order, the original value of the register can be restored from the history file. A similar technique is used for autoincrement and autodecrement addressing on processors such as VAXes. Another approach, the *future file*, proposed by Smith and Pleszkun (1988), keeps the newer value of a register; when all earlier instructions have completed, the main register file is updated from the future file. On an exception, the main register file has the precise values for the interrupted state. In Chapter 3, we will see another approach that is needed to support speculation, a method of executing instructions before we know the outcome of previous branches.

A third technique in use is to allow the exceptions to become somewhat imprecise, but to keep enough information so that the trap-handling routines can create a precise sequence for the exception. This means knowing what operations were in the pipeline and their PCs. Then, after handling the exception, the software finishes any instructions that precede the latest instruction completed, and the sequence can restart. Consider the following worst-case code sequence:

Instruction$_1$—A long-running instruction that eventually interrupts execution.

Instruction$_2$, … , Instruction$_{n-1}$—A series of instructions that are not completed.

Instruction$_n$—An instruction that is finished.

Given the PCs of all the instructions in the pipeline and the exception return PC, the software can find the state of instruction1 and instruction$_n$. Because instruction$_n$ has completed, we will want to restart execution at instruction$_{n+1}$. After handling the exception, the software must simulate the execution of instruction$_1$, … , instruction$_{n-1}$. Then we can return from the exception and restart at instruction$_{n+1}$. The complexity of executing these instructions properly by the handler is the major difficulty of this scheme.

There is an important simplification for simple RISC V-like pipelines: If instruction$_2$, … , instruction$_n$ are all integer instructions, we know that if instruction$_n$ has completed then all of instruction$_2$, … , instruction$_{n-1}$ have also completed. Thus, only FP operations need to be handled. To make this scheme tractable, the number of floating-point instructions that can be overlapped in execution can be limited. For example, if we only overlap two instructions, then only the interrupting instruction need be completed by software. This restriction may reduce the potential throughput if the FP pipelines are deep or if there are a significant number of FP functional units. This approach is used in some SPARC implementations to allow overlap of floating-point and integer operations.

The final technique is a hybrid scheme that allows the instruction issue to continue only if it is certain that all the instructions before the issuing instruction will complete without causing an exception. This guarantees that when an exception occurs, no instructions after the interrupting one will be completed and all of the instructions before the interrupting one can be completed. This sometimes means stalling the processor to maintain precise exceptions. To make this scheme work, the floating-point functional units must determine if an exception is possible early in the EX stage (in the first 3 clock cycles in the RISC V pipeline), so as to prevent further instructions from completing. This scheme is used in the MIPS R2000/3000, the R4000, and the Intel Pentium. It is discussed further in Appendix J.

Performance of a Simple RISC V FP Pipeline

The RISC V FP pipeline of Figure C.30 on page C.48 can generate both structural stalls for the divide unit and stalls for RAW hazards (it also can have WAW hazards, but this rarely occurs in practice). Figure C.34 shows the number of stall cycles for each type of floating-point operation on a per-instance basis (i.e., the first bar for each FP benchmark shows the number of FP result stalls for each FP add, subtract, or convert). As we might expect, the stall cycles per operation track the latency of the FP operations, varying from 46% to 59% of the latency of the functional unit.

Figure C.35 gives the complete breakdown of integer and FP stalls for five SPECfp benchmarks. There are four classes of stalls shown: FP result stalls, FP compare stalls, load and branch delays, and FP structural delays. Branch delay stalls, which would be small with a one cycle delay and even a modest branch predictor, are not included. The total number of stalls per instruction varies from 0.65 to 1.21.

C.6 Putting It All Together: The MIPS R4000 Pipeline

In this section, we look at the pipeline structure and performance of the MIPS R4000 processor family, which includes the 4400. The MIPS architecture and RISC V are very similar, differing only in a few instructions, including a delayed branch in the MIPS ISA. The R4000 implements MIPS64 but uses a deeper pipeline than that of our five-stage design both for integer and FP programs. This deeper pipeline allows it to achieve higher clock rates by decomposing the five-stage integer pipeline into eight stages. Because cache access is particularly time critical, the extra pipeline stages come from decomposing the memory access. This type of deeper pipelining is sometimes called *superpipelining*.

Figure C.36 shows the eight-stage pipeline structure using an abstracted version of the data path. Figure C.37 shows the overlap of successive instructions in the pipeline. Notice that, although the instruction and data memory occupy multiple cycles, they are fully pipelined, so that a new instruction can start on every clock.

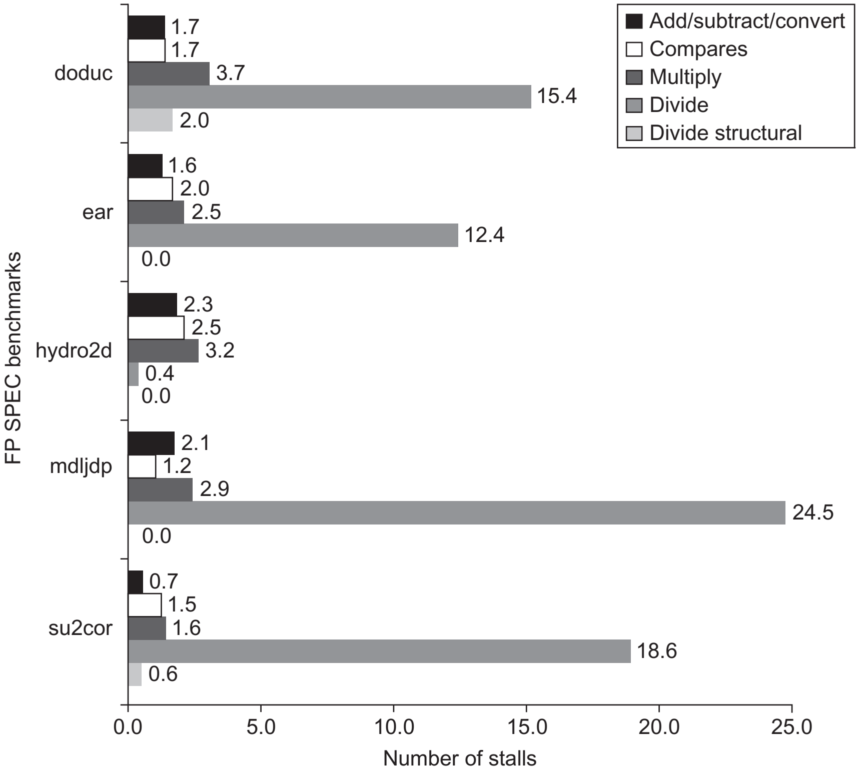


Figure C.34 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (three cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, because the divide frequency is low.

In fact, the pipeline uses the data before the cache hit detection is complete; Chapter 3 discusses how this can be done in more detail.

The function of each stage is as follows:

- IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, complete instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and instruction cache hit detection.

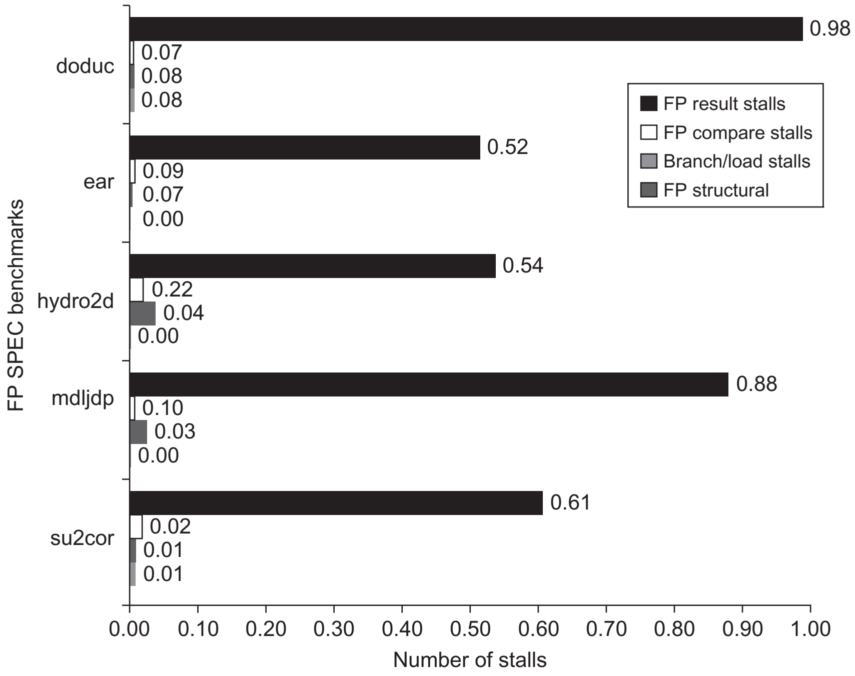


Figure C.35 The stalls occurring for the a simple RISC V FP pipeline for five of the SPEC89 FP benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc. Branch stalls are not accounted for, but would be small.

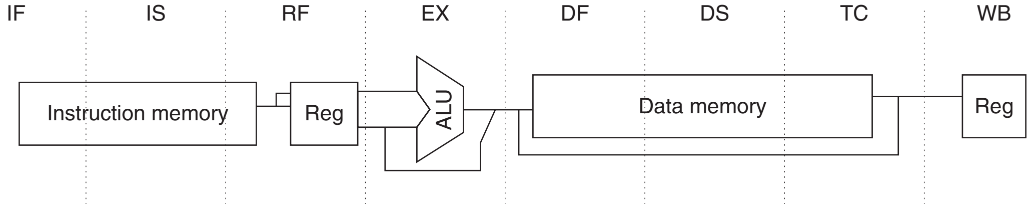


Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

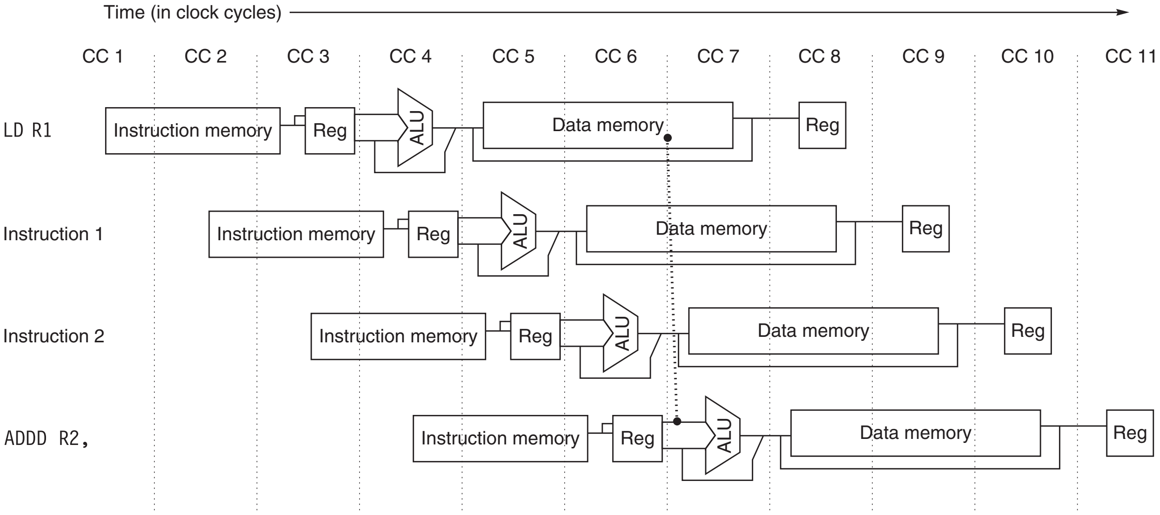


Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

- EX—Execution, which includes effective address calculation, ALU operation, and branch-target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, to determine whether the data cache access hit.
- WB—Write-back for loads and register-register operations.

In addition to substantially increasing the amount of forwarding required, this longer-latency pipeline increases both the load and branch delays. Figure C.37 shows that load delays are two cycles, because the data value is available at the end of DS. Figure C.38 shows the shorthand pipeline schedule when a use immediately follows a load. It shows that forwarding is required for the result of a load instruction to a destination that is three or four cycles later.

Figure C.39 shows that the basic branch delay is three cycles, because the branch condition is computed during EX. The MIPS architecture has a single-cycle delayed branch. The R4000 uses a predicted-not-taken strategy for the remaining two cycles of the branch delay. As Figure C.40 shows, untaken branches are simply one-cycle delayed branches, while taken branches have a one-cycle delay slot followed by two idle cycles. The instruction set provides a branch-likely instruction, which we described earlier and which helps in filling the branch delay slot.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`ld x1,...`	IF	IS	RF	EX	DF	DS	TC	WB	
`add x2,x1,...`		IF	IS	RF	Stall	Stall	EX	DF	DS
`sub x3,x1,...`			IF	IS	Stall	Stall	RF	EX	DF
`or x4,x1,...`				IF	Stall	Stall	IS	RF	EX

Figure C.38 A load instruction followed by an immediate use results in a x1 stall. Normal forwarding paths can be used after two cycles, so the `add` and `sub` get the value by forwarding after the stall. The `or` instruction gets the value from the register file. Because the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are three or four cycles after the load.

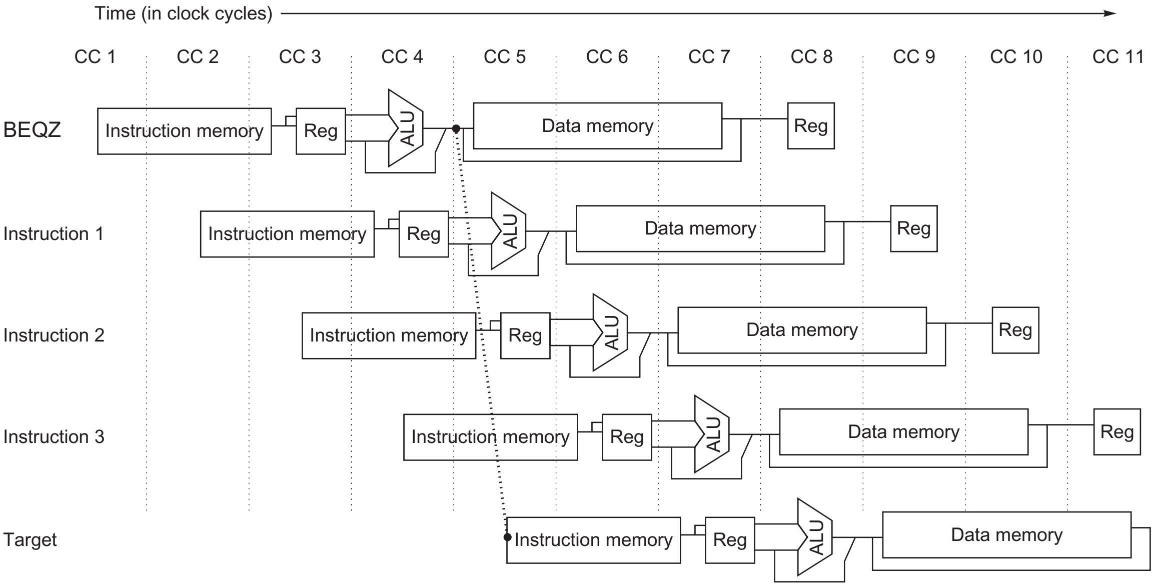

Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
CC 10
CC 11
BEQZ
Instruction memory
Reg
ALU
Data memory
Reg
Instruction 1
Instruction memory
Reg
ALU
Data memory
Reg
Instruction 2
Instruction memory
Reg
ALU
Data memory
Reg
Instruction 3
Instruction memory
Reg
ALU
Data memory
Reg
Target
Instruction memory
Reg
ALU
Data memory

Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

Pipeline interlocks enforce both the x1 branch stall penalty on a taken branch and any data hazard stall that arises from use of a load result. After the R4000, all implementations of MIPS processor made use of dynamic branch prediction.

In addition to the increase in stalls for loads and branches, the deeper pipeline increases the number of levels of forwarding for ALU operations. In our RISC V five-stage pipeline, forwarding between two register-register ALU instructions could happen from the ALU/MEM or the MEM/WB registers. In the R4000 pipeline, there are four possible sources for an ALU bypass: EX/DF, DF/DS, DS/TC, and TC/WB.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			Stall	Stall	Stall	Stall	Stall	Stall	Stall
Stall				Stall	Stall	Stall	Stall	Stall	Stall
Branch target					IF	IS	RF	EX	DF
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
Branch instruction + 3				IF	IS	RF	EX	DF	DS

Figure C.40 A taken branch, shown in the top portion of the figure, has a one-cycle delay slot followed by a x1 stall, while an untaken branch, shown in the bottom portion, has simply a one-cycle delay slot. The branch instruction can be an ordinary delayed branch or a branch-likely, which cancels the effect of the instruction in the delay slot if the branch is untaken.

The Floating-Point Pipeline

The R4000 floating-point unit consists of three functional units: a floating-point divider, a floating-point multiplier, and a floating-point adder. The adder logic is used on the final step of a multiply or divide. Double-precision FP operations can take from 2 cycles (for a negate) up to 112 cycles (for a square root). In addition, the various units have different initiation rates. The FP functional unit can be thought of as having eight different stages, listed in Figure C.41; these stages are combined in different orders to execute various FP operations.

There is a single copy of each of these stages, and various instructions may use a stage zero or more times and in different orders. Figure C.42 shows the latency, initiation rate, and pipeline stages used by the most common double-precision FP operations.

Stage	**Functional unit**	**Description**
A	FP adder	Mantissa add stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Figure C.41 The eight stages used in the R4000 floating-point pipelines.

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U, E+M, M, M, M, N, N+A, R
Divide	36	35	U, A, R, D^{28}, D+A, D+R, D+A, D+R, A, R
Square root	112	111	U, E, $(A+R)^{108}$, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Figure C.42 The latencies and initiation intervals for the FP operations initiation intervals for the FP operations both depend on the FP unit stages that a given operation must use. The latency values assume that the destination instruction is an FP operation; the latencies are one cycle less when the destination is a store. The pipe stages are shown in the order in which they are used for any operation. The notation S+A indicates a clock cycle in which both the S and A stages are used. The notation D^{28} indicates that the D stage is used 28 times in a row.

From the information in Figure C.42, we can determine whether a sequence of different, independent FP operations can issue without stalling. If the timing of the sequence is such that a conflict occurs for a shared pipeline stage, then a stall will be needed. Figures C.43–C.46 show four common possible two-instruction sequences: a multiply followed by an add, an add followed by a multiply, a divide followed by an add, and an add followed by a divide. The figures show all the interesting starting positions for the second instruction and whether that second instruction will issue or stall for each position. Of course, there could be three instructions active, in which case the possibilities for stalls are much higher and the figures more complex.

Performance of the R4000 Pipeline

In this section, we examine the stalls that occur for the SPEC92 benchmarks when running on the R4000 pipeline structure. There are four major causes of pipeline stalls or losses:

1. *Load stalls*—Delays arising from the use of a load result one or two cycles after the load
2. *Branch stalls*—Two-cycle stalls on every taken branch plus unfilled or canceled branch delay slots. The version of the MIPS instruction set implemented in the R4000 supports instructions that predict a branch at compile time and cause the instruction in the branch delay slot to be canceled when the branch behavior differs from the prediction. This makes it easier to fill branch delay slots.
3. *FP result stalls*—Stalls because of RAW hazards for an FP operand
4. *FP structural stalls*—Delays because of issue restrictions arising from conflicts for functional units in the FP pipeline

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Multiply	Issue	U	E+M	M	M	M	N	**N+A**	**R**					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	**A+R**	**R+S**					
	Stall						U	**S+A**	**A+R**	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

Figure C.43 An FP multiply issued at clock 0 is followed by a single FP add issued between clocks 1 and 7. The second column indicates whether an instruction of the specified type stalls when it is issued *n* cycles later, where *n* is the clock cycle number in which the U stage of the second instruction occurs. The stage or stages that cause a stall are in bold. Note that this table deals with only the interaction between the multiply and *one* add issued between clocks 1 and 7. In this case, the add will stall if it is issued four or five cycles after the multiply; otherwise, it issues without stalling. Notice that the add will be stalled for two cycles if it issues in cycle 4 because on the next clock cycle it will still conflict with the multiply; if, however, the add issues in cycle 5, it will stall for only 1 clock cycle, because that will eliminate the conflicts.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	A+R	R+S									
Multiply	Issue		U	E+M	M	M	M	N	N+A	R				
	Issue			U	M	M	M	M	N	N+A	R			

Figure C.44 A multiply issuing after an add can always proceed without stalling, because the shorter instruction clears the shared pipeline stages before the longer instruction reaches them.

Figure C.47 shows the pipeline CPI breakdown for the R4000 pipeline for the 10 SPEC92 benchmarks. Figure C.48 shows the same data but in tabular form.

From the data in Figures C.47 and C.48, we can see the penalty of the deeper pipelining. The R4000's pipeline has much longer branch delays than the classic five-stage pipeline. The longer branch delay substantially increases the cycles spent on branches, especially for the integer programs with a higher branch frequency. This is the reason that almost all subsequent processors with moderate to deep pipelines (8–16 stages are typical today) employ dynamic branch predictors.

		Clock cycle											
Operation	**Issue/stall**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Divide	Issued in cycle 0…	D	D	D	D	D	**D+A**	**D+R**	**D+A**	**D+R**	**A**	**R**	
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Stall				U	S+A	**A+R**	**R+S**					
	Stall					U	**S+A**	**A+R**	R+S				
	Stall						U	S+A	**A+R**	**R+S**			
	Stall							U	**S+A**	**A+R**	R+S		
	Stall								U	S+A	**A+R**	**R+S**	
	Stall									U	**S+A**	**A+R**	R+S
	Issue										U	S+A	A+R
	Issue											U	S+A
	Issue												U

Figure C.45 An FP divide can cause a stall for an add that starts near the end of the divide. The divide starts at cycle 0 and completes at cycle 35; the last 10 cycles of the divide are shown. Because the divide makes heavy use of the rounding hardware needed by the add, it stalls an add that starts in any of cycles 28–33. Notice that the add starting in cycle 28 will be stalled until cycle 36. If the add started right after the divide, it would not conflict, because the add could complete before the divide needed the shared stages, just as we saw in Figure C.44 for a multiply and add. As in the earlier figure, this example assumes *exactly* one add that reaches the U stage between clock cycles 26 and 35.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	**A+R**	**R+S**									
Divide	Stall		U	**A**	**R**	D	D	D	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D	D	D	D
	Issue				U	A	R	D	D	D	D	D	D	D

Figure C.46 A double-precision add is followed by a double-precision divide. If the divide starts one cycle after the add, the divide stalls, but after that there is no conflict.

An interesting effect observed in the FP programs is that the latency of the FP functional units leads to more result stalls than the structural hazards, which arise both from the initiation interval limitations and from conflicts for functional units from different FP instructions. Thus, reducing the latency of FP operations should be the first target, rather than more pipelining or replication of the functional units. Of course, reducing the latency would probably increase the structural stalls, because many potential structural stalls are hidden behind data hazards.

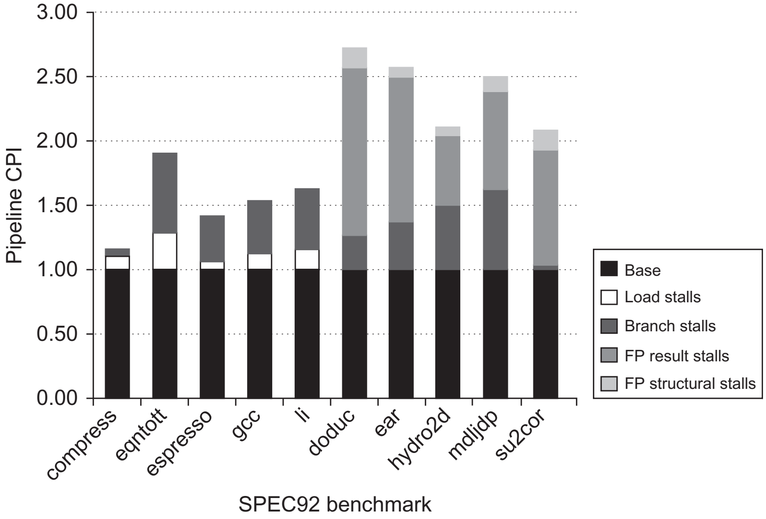


Figure C.47 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The left-most five programs are integer programs, and branch delays are the major CPI contributor for these. The right-most five programs are FP, and FP result stalls are the major contributor for these. Figure C.48 shows the numbers used to construct this plot.

Benchmark	Pipeline CPI	Load stalls	Branch stalls	FP result stalls	FP structural stalls
Compress	1.20	0.14	0.06	0.00	0.00
Eqntott	1.88	0.27	0.61	0.00	0.00
Espresso	1.42	0.07	0.35	0.00	0.00
Gcc	1.56	0.13	0.43	0.00	0.00
Li	1.64	0.18	0.46	0.00	0.00
Integer average	1.54	0.16	0.38	0.00	0.00
Doduc	2.84	0.01	0.22	1.39	0.22
Mdljdp2	2.66	0.01	0.31	1.20	0.15
Ear	2.17	0.00	0.46	0.59	0.12
Hydro2d	2.53	0.00	0.62	0.75	0.17
Su2cor	2.18	0.02	0.07	0.84	0.26
FP average	2.48	0.01	0.33	0.95	0.18
Overall average	2.00	0.10	0.36	0.46	0.09

Figure C.48 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

C.7 Cross-Cutting Issues

RISC Instruction Sets and Efficiency of Pipelining

We have already discussed the advantages of instruction set simplicity in building pipelines. Simple instruction sets offer another advantage: they make it easier to schedule code to achieve efficiency of execution in a pipeline. To see this, consider a simple example: suppose we need to add two values in memory and store the result back to memory. In some sophisticated instruction sets this will take only a single instruction; in others, it will take two or three. A typical RISC architecture would require four instructions (two loads, an add, and a store). These instructions cannot be scheduled sequentially in most pipelines without intervening stalls.

With a RISC instruction set, the individual operations are separate instructions and may be individually scheduled either by the compiler (using the techniques we discussed earlier and more powerful techniques discussed in Chapter 3) or using dynamic hardware scheduling techniques (which we discuss next and in further detail in Chapter 3). These efficiency advantages, coupled with the greater ease of implementation, appear to be so significant that almost all recent pipelined implementations of complex instruction sets actually translate their complex instructions into simple RISC-like operations, and then schedule and pipeline those operations. All recent Intel processors use this approach, and it is also used in ARM processors for some of the more complex instructions.

Dynamically Scheduled Pipelines

Simple pipelines fetch an instruction and issue it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. Forwarding logic reduces the effective pipeline latency so that certain dependences do not result in hazards. If there is an unavoidable hazard, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result). No new instructions are fetched or issued until the dependence is cleared. To overcome these performance losses, the compiler can attempt to schedule instructions to avoid the hazard; this approach is called *compiler* or *static scheduling*.

Several early processors used another approach, called *dynamic scheduling*, whereby the hardware rearranges the instruction execution to reduce the stalls. This section offers a simpler introduction to dynamic scheduling by explaining the scoreboarding technique of the CDC 6600. Some readers will find it easier to read this material before plunging into the more complicated Tomasulo scheme, and the speculation approaches that extend it, both of which are covered in Chapter 3.

All the techniques discussed in this appendix so far use in-order instruction issue, which means that if an instruction is stalled in the pipeline, no later instructions can proceed. With in-order issue, if two instructions have a hazard between them, the pipeline will stall, even if there are later instructions that are independent and would not stall.

In the RISC V pipeline developed earlier, both structural and data hazards were checked during instruction decode (ID): when an instruction could execute properly, it was issued from ID. To allow an instruction to begin execution as soon as its operands are available, even if a predecessor is stalled, we must separate the issue process into two parts: checking the structural hazards and waiting for the absence of a data hazard. We decode and issue instructions in order; however, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do *out-of-order execution*, which implies *out-of-order completion*. To implement out-of-order execution, we must split the ID pipe stage into two stages:

1. *Issue*—Decode instructions, check for structural hazards.
2. *Read operands*—Wait until no data hazards, then read operands.

The IF stage proceeds the issue stage, and the EX stage follows the read operands stage, just as in the RISC V pipeline. As in the RISC V floating-point pipeline, execution may take multiple cycles, depending on the operation. Thus, we may need to distinguish when an instruction *begins execution* and when it *completes execution*; between the two times, the instruction is *in execution*. This allows multiple instructions to be in execution at the same time. In addition to these changes to the pipeline structure, we will also change the functional unit design by varying the number of units, the latency of operations, and the functional unit pipelining so as to better explore these more advanced pipelining techniques.

Dynamic Scheduling With a Scoreboard

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability.

Before we see how scoreboarding could be used in the RISC V pipeline, it is important to observe that WAR hazards, which did not exist in the RISC V floating-point or integer pipelines, may arise when instructions execute out of order. For example, consider the following code sequence:

```
fdiv.d     f0,f2,f4
fadd.d     f10,f0,f8
fsub.d     f8,f8,f14
```

There is an potential WAR hazard between the `fadd.d` and the `fsub.d`: If the pipeline executes the `fsub.d` before the `fadd.d`, it will violate yield incorrect execution. Likewise, the pipeline must avoid WAW hazards (e.g.,as would occur if the destination of the `fsub.d` were `f10`). As we will see, both these hazards are avoided in a scoreboard by stalling the later instruction involved in the hazard.

The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection. Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously. This can be achieved with multiple functional units, with pipelined functional units, or with both. Because these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

The CDC 6600 had 16 separate functional units, including 4 floating-point units, 5 units for memory references, and 7 units for integer operations. On a processor for the RISC V architecture, scoreboards make sense primarily on the floating-point unit because the latency of the other functional units is very small. Let's assume that there are two multipliers, one adder, one divide unit, and a single integer unit for all memory references, branches, and integer operations. Although this example is simpler than the CDC 6600, it is sufficiently powerful to demonstrate the principles without having a mass of detail or needing very long examples. Because both RISC V and the CDC 6600 are load-store architectures, the techniques are nearly identical for the two processors. Figure C.49 shows what the processor looks like.

Every instruction goes through the scoreboard, where a record of the data dependences is constructed; this step corresponds to instruction issue and replaces part of the ID step in the RISC V pipeline. The scoreboard then determines when the instruction can read its operands and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction *can* execute. The scoreboard also controls when an instruction can write its result into the destination register. Thus, all hazard detection and resolution are centralized in the scoreboard. We will see a picture of the scoreboard later (Figure C.49 on page C.68), but first we need to understand the steps in the issue and execution segment of the pipeline.

Each instruction undergoes four steps in executing. (Because we are concentrating on the FP operations, we will not consider a step for memory access.) Let's first examine the steps informally and then look in detail at how the scoreboard keeps the necessary information that determines when to progress from one step to the next. The four steps, which replace the ID, EX, and WB steps in the standard RISC V pipeline, are as follows:

1. *Issue*—If a functional unit for the instruction is free and no other active instruction has the same destination register, the scoreboard issues the instruction to the functional unit and updates its internal data structure. This step replaces a portion of the ID step in the RISC V pipeline. By ensuring that no other active

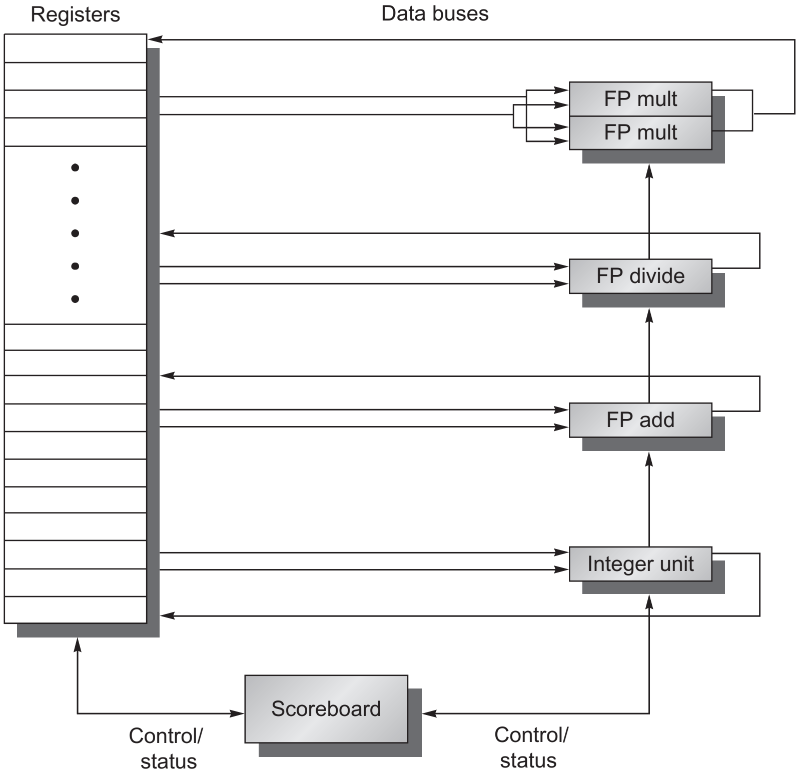


Figure C.49 The basic structure of a RISC V processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. We will explore scoreboarding and its extensions in more detail in Chapter 3.

functional unit wants to write its result into the destination register, we guarantee that WAW hazards cannot be present. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared. When the issue stage stalls, it causes the buffer between instruction fetch and issue to fill; if the buffer is a single entry, instruction fetch stalls immediately. If the buffer is a queue with multiple instructions, it stalls when the queue fills.

2. *Read operands*—The scoreboard monitors the availability of the source operands. A source operand is available if no earlier issued active instruction is going to write it. When the source operands are available, the scoreboard tells

the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order. This step, together with issue, completes the function of the ID step in the simple RISC V pipeline.

3. *Execution*—The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution. This step replaces the EX step in the RISC V pipeline and takes multiple cycles in the RISC V FP pipeline.
4. *Write result*—Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards and stalls the completing instruction, if necessary.

A WAR hazard exists if there is a code sequence like our earlier example with `fadd.d` and `fsub.d` that both use `f8`. In that example, we had the code

```
fdiv.d     f0,f2,f4
fadd.d     f10,f0,f8
fsub.d     f8,f8,f14
```

`fadd.d` has a source operand `f8`, which is the same register as the destination of `fsub.d`. But `fadd.d` actually depends on an earlier instruction. The scoreboard will still stall the `fsub.d` in its write result stage until `fadd.d` reads its operands. In general, then, a completing instruction cannot be allowed to write its results when:

- There is an instruction that has not read its operands that precedes (i.e., in order of issue) the completing instruction, and
- One of the operands is the same register as the result of the completing instruction.

If this WAR hazard does not exist, or when it clears, the scoreboard tells the functional unit to store its result to the destination register. This step replaces the WB step in the simple RISC V pipeline.

At first glance, it might appear that the scoreboard will have difficulty separating RAW and WAR hazards.

Because the operands for an instruction are read only when both operands are available in the register file, this scoreboard does not take advantage of forwarding. Instead, registers are only read when they are both available. This is not as large a penalty as you might initially think. Unlike our simple pipeline of earlier, instructions will write their result into the register file as soon as they complete execution (assuming no WAR hazards), rather than wait for a statically assigned write slot that may be several cycles away. The effect reduces the pipeline latency and the benefits of forwarding. There is still one additional cycle of latency that arises because the write result and read operand stages cannot overlap. We would need additional buffering to eliminate this overhead.

Based on its own data structure, the scoreboard controls the instruction progression from one step to the next by communicating with the functional units. There is a small complication, however. There are only a limited number of source operand buses and result buses to the register file, which represents a structural hazard. The scoreboard must guarantee that the number of functional units allowed to proceed into steps 2 and 4 does not exceed the number of buses available. We will not go into further detail on this, other than to mention that the CDC 6600 solved this problem by grouping the 16 functional units together into four groups and supplying a set of buses, called *data trunks*, for each group. Only one unit in a group could read its operands or write its result during a clock.

C.8 Fallacies and Pitfalls

Pitfall *Unexpected execution sequences may cause unexpected hazards.*

At first glance, WAW hazards look like they should never occur in a code sequence because no compiler would ever generate two writes to the same register without an intervening read, but they can occur when the sequence is unexpected. For example, consider a long running floating point divide that causes a trap. If the trap routine writes the same register as the divide early on, it may cause a WAW hazard, if it writes the register before the divide completes. Hardware or software must avoid this possibility.

Pitfall *Extensive pipelining can impact other aspects of a design, leading to overall worse cost-performance.*

The best example of this phenomenon comes from two implementations of the VAX, the 8600 and the 8700. When the 8600 was initially delivered, it had a cycle time of 80 ns. Subsequently, a redesigned version, called the 8650, with a 55 ns clock was introduced. The 8700 has a much simpler pipeline that operates at the microinstruction level, yielding a smaller processor with a faster clock cycle of 45 ns. The overall outcome is that the 8650 has a CPI advantage of about 20%, but the 8700 has a clock rate that is about 20% faster. Thus, the 8700 achieved the same performance with much less hardware.

Pitfall *Evaluating dynamic or static scheduling on the basis of unoptimized code.*

Unoptimized code—containing redundant loads, stores, and other operations that might be eliminated by an optimizer—is much easier to schedule than "tight" optimized code. This holds for scheduling both control delays (with delayed branches) and delays arising from RAW hazards. In gcc running on an R3000, which has a pipeline almost identical to that of Section C.1, the frequency of idle clock cycles increases by 18% from the unoptimized and scheduled code to the optimized and scheduled code. Of course, the optimized program is much faster, because it has fewer instructions. To fairly evaluate a compile-time scheduler or runtime dynamic scheduling, you must use optimized code, because in the real system you will derive good performance from other optimizations in addition to scheduling.

C.9 Concluding Remarks

At the beginning of the 1980s, pipelining was a technique reserved primarily for supercomputers and large multimillion-dollar mainframes. By the mid-1980s, the first pipelined microprocessors appeared and helped transform the world of computing, allowing microprocessors to bypass minicomputers in performance and eventually to take on and outperform mainframes. By the early 1990s, high-end embedded microprocessors embraced pipelining, and desktops were headed toward the use of the sophisticated dynamically scheduled, multiple-issue approaches discussed in Chapter 3. The material in this appendix, which was considered reasonably advanced for graduate students when this text first appeared in 1990, is now considered basic undergraduate material and can be found in processors that cost less than $1!

C.10 Historical Perspective and References

Section M.5 (available online) features a discussion on the development of pipelining and instruction-level parallelism covering both this appendix and the material in Chapter 3. We provide numerous references for further reading and exploration of these topics.

Updated Exercises by Diana Franklin

C.1 [15/15/15/15/25/10/15] < A.2 > Use the following code fragment:

```
Loop:  ld     x1,0(x2)     ;load x1 from address 0+x2
       addi   x1,x1,1      ;x1=x1+1
       sd     x1,0,(x2)    ;store x1 at address 0+x2
       addi   x2,x2,4      ;x2=x2+4
       sub    x4,x3,x2     ;x4=x3-x2
       bnez   x4,Loop      ;branch to Loop if x4!= 0
```

Assume that the initial value of x3 is x2 + 396.

a. [15] < C.2 > Data hazards are caused by data dependences in the code. Whether a dependency causes a hazard depends on the machine implementation (i.e., number of pipeline stages). List all of the data dependences in the code above. Record the register, source instruction, and destination instruction; for example, there is a data dependency for register x1 from the `ld` to the `addi`.

b. [15] < C.2 > Show the timing of this instruction sequence for the 5-stage RISC pipeline without any forwarding or bypassing hardware but assuming that a register read and a write in the same clock cycle "forwards" through the register file, as between the `add` and `or` shown in Figure C.5. Use a pipeline timing chart like that in Figure C.8. Assume that the branch is handled by flushing the pipeline. If all memory references take 1 cycle, how many cycles does this loop take to execute?

c. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.8. Assume that the branch is handled by predicting it as not taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

d. [15] <C.2> Show the timing of this instruction sequence for the 5-stage RISC pipeline with full forwarding and bypassing hardware, as shown in Figure C.6. Use a pipeline timing chart like that shown in Figure C.8. Assume that the branch is handled by predicting it as taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

e. [25] <C.2> High-performance processors have very deep pipelines—more than 15 stages. Imagine that you have a 10-stage pipeline in which every stage of the 5-stage pipeline has been split in two. The only catch is that, for data forwarding, data are forwarded from the end of a *pair of stages* to the beginning of the two stages where they are needed. For example, data are forwarded from the output of the second execute stage to the input of the first execute stage, still causing a 1-cycle delay. Show the timing of this instruction sequence for the 10-stage RISC pipeline with full forwarding and bypassing hardware. Use a pipeline timing chart like that shown in Figure C.8 (but with stages labeled IF1, IF2, ID1, etc.). Assume that the branch is handled by predicting it as taken. If all memory references take 1 cycle, how many cycles does this loop take to execute?

f. [10] <C.2> Assume that in the 5-stage pipeline, the longest stage requires 0.8 ns, and the pipeline register delay is 0.1 ns. What is the clock cycle time of the 5-stage pipeline? If the 10-stage pipeline splits all stages in half, what is the cycle time of the 10-stage machine?

g. [15] <C.2> Using your answers from parts (d) and (e), determine the cycles per instruction (CPI) for the loop on a 5-stage pipeline and a 10-stage pipeline. Make sure you count only from when the first instruction reaches the write-back stage to the end. Do not count the start-up of the first instruction. Using the clock cycle time calculated in part (f), calculate the average instruction execute time for each machine.

C.2 [15/15] <C.2> Suppose the branch frequencies (as percentages of all instructions) are as follows:

Conditional branches	15%
Jumps and calls	1%
Taken conditional branches	60% are taken

a. [15] <C.2> We are examining a four-stage pipeline where the branch is resolved at the end of the second cycle for unconditional branches and at the end of the third cycle for conditional branches. Assuming that only the first pipe stage can always be completed independent of whether the branch is taken and ignoring other pipeline stalls, how much faster would the machine be without any branch hazards?

b. [15] <C.2> Now assume a high-performance processor in which we have a 15-deep pipeline where the branch is resolved at the end of the fifth cycle for unconditional branches and at the end of the tenth cycle for conditional branches. Assuming that only the first pipe stage can always be completed independent of whether the branch is taken and ignoring other pipeline stalls, how much faster would the machine be without any branch hazards?

C.3 [5/15/10/10] <C.2> We begin with a computer implemented in single-cycle implementation. When the stages are split by functionality, the stages do not require exactly the same amount of time. The original machine had a clock cycle time of 7 ns. After the stages were split, the measured times were IF, 1 ns; ID, 1.5 ns; EX, 1 ns; MEM, 2 ns; and WB, 1.5 ns. The pipeline register delay is 0.1 ns.

a. [5] <C.2> What is the clock cycle time of the 5-stage pipelined machine?

b. [15] <C.2> If there is a stall every four instructions, what is the CPI of the new machine?

c. [10] <C.2> What is the speedup of the pipelined machine over the single-cycle machine?

d. [10] <C.2> If the pipelined machine had an infinite number of stages, what would its speedup be over the single-cycle machine?

C.4 [15] <C.1, C.2> A reduced hardware implementation of the classic five-stage RISC pipeline might use the EX stage hardware to perform a branch instruction comparison and then not actually deliver the branch target PC to the IF stage until the clock cycle in which the branch instruction reaches the MEM stage. Control hazard stalls can be reduced by resolving branch instructions in ID, but improving performance in one respect may reduce performance in other circumstances. Write a small snippet of code in which calculating the branch in the ID stage causes a data hazard, even with data forwarding.

C.5 [12/13/20/20/15/15] <C.2, C.3> For these problems, we will explore a pipeline for a register-memory architecture. The architecture has two instruction formats: a register-register format and a register-memory format. There is a single-memory addressing mode (offset + base register). There is a set of ALU operations with the format:

```
ALUop Rdest, Rsrc1, Rsrc2
```

or

```
ALUop Rdest, Rsrc1, MEM
```

where the ALUop is one of the following: add, subtract, AND, OR, load (Rsrc1 ignored), or store. Rsrc or Rdest are registers. MEM is a base register and offset pair. Branches use a full compare of two registers and are PC relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The pipeline structure, similar to that used in the VAX 8700 micropipeline (Clark, 1987), is

IF	**RF**	**ALU1**	**MEM**	**ALU2**	**WB**					
	IF	RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU cycle is used for operations and branch comparison. RF is both a decode and register-fetch cycle. Assume that when a register read and a register write of the same register occur in the same clock, the write data are forwarded.

a. [12] <C.2> Find the number of adders needed, counting any adder or incrementer; show a combination of instructions and pipe stages that justify this answer. You need only give one combination that maximizes the adder count.

b. [13] <C.2> Find the number of register read and write ports and memory read and write ports required. Show that your answer is correct by showing a combination of instructions and pipeline stage indicating the instruction and the number of read ports and write ports required for that instruction.

c. [20] <C.3> Determine any data forwarding for any ALUs that will be needed. Assume that there are separate ALUs for the ALU1 and ALU2 pipe stages. Put in all forwarding among ALUs necessary to avoid or reduce stalls. Show the relationship between the two instructions involved in forwarding using the format of the table in Figure C.23 but ignoring the last two columns. Be careful to consider forwarding across an intervening instruction—for example,

```
add            x1, ...
any instruction
add            ..., x1, ...
```

d. [20] <C.3> Show all of the data forwarding requirements necessary to avoid or reduce stalls when either the source or destination unit is not an ALU. Use the same format as in Figure C.23, again ignoring the last two columns. Remember to forward to and from memory references.

e. [15] < C.3 > Show all the remaining hazards that involve at least one unit other than an ALU as the source or destination unit. Use a table like that shown in Figure C.25, but replace the last column with the lengths of the hazards.

f. [15] < C.2 > Show all control hazards by example and state the length of the stall. Use a format like that shown in Figure C.11, labeling each example.

C.6 [12/13/13/15/15] < C.1, C.2, C.3 > We will now add support for register-memory ALU operations to the classic five-stage RISC pipeline. To offset this increase in complexity, *all* memory addressing will be restricted to register indirect (i.e., all addresses are simply a value held in a register; no offset or displacement may be added to the register value). For example, the register-memory instruction `add x4, x5, (x1)` means add the contents of register x5 to the contents of the memory location with address equal to the value in register x1 and put the sum in register x4. Register-register ALU operations are unchanged. The following items apply to the integer RISC pipeline:

a. [12] < C.1 > List a rearranged order of the five traditional stages of the RISC pipeline that will support register-memory operations implemented exclusively by register indirect addressing.

b. [13] < C.2, C.3 > Describe what new forwarding paths are needed for the rearranged pipeline by stating the source, destination, and information transferred on each needed new path.

c. [13] < C.2, C.3 > For the reordered stages of the RISC pipeline, what new data hazards are created by this addressing mode? Give an instruction sequence illustrating each new hazard.

d. [15] < C.3 > List all of the ways that the RISC pipeline with register-memory ALU operations can have a different instruction count for a given program than the original RISC pipeline. Give a pair of specific instruction sequences, one for the original pipeline and one for the rearranged pipeline, to illustrate each way.

e. [15] < C.3 > Assume that all instructions take 1 clock cycle per stage. List all of the ways that the register-memory RISC V can have a different CPI for a given program as compared to the original RISC V pipeline.

C.7 [10/10] < C.3 > In this problem, we will explore how deepening the pipeline affects performance in two ways: faster clock cycle and increased stalls due to data and control hazards. Assume that the original machine is a 5-stage pipeline with a 1 ns clock cycle. The second machine is a 12-stage pipeline with a 0.6 ns clock cycle. The 5-stage pipeline experiences a stall due to a data hazard every five instructions, whereas the 12-stage pipeline experiences three stalls every eight instructions. In addition, branches constitute 20% of the instructions, and the misprediction rate for both machines is 5%.

a. [10] < C.3 > What is the speedup of the 12-stage pipeline over the 5-stage pipeline, taking into account only data hazards?

b. [10] <C.3> If the branch mispredict penalty for the first machine is 2 cycles but the second machine is 5 cycles, what are the CPIs of each, taking into account the stalls due to branch mispredictions?

C.8 [15] <C.5> Construct a table like that shown in Figure C.21 to check for WAW stalls in the RISC V FP pipeline of Figure C.30. Do not consider FP divides.

C.9 [20/22/22] <C.4, C.6> In this exercise, we will look at how a common vector loop runs on statically and dynamically scheduled versions of the RISC V pipeline. The loop is the so-called DAXPY loop (discussed extensively in Appendix G) and the central operation in Gaussian elimination. The loop implements the vector operation $Y = a*X + Y$ for a vector of length 100. Here is the MIPS code for the loop:

```
foo:   fld      f2, 0(x1)     ; load X(i)
       fmul.d   f4, f2, f0    ; multiply a*X(i)
       fld      f6, 0(x2)     ; load Y(i)
       fadd.d   f6, f4, f6    ; add a*X(i) + Y(i)
       fsd      0(x2), f6     ; store Y(i)
       addi     x1, x1, 8     ; increment X index
       addi     x2, x2, 8     ; increment Y index
       sltiu    x3, x1, done  ; test if done
       bnez     x3, foo       ; loop if not done
```

For parts (a) to (c), assume that integer operations issue and complete in 1 clock cycle (including loads) and that their results are fully bypassed. You will use the FP latencies (only) shown in Figure C.29, but assume that the FP unit is fully pipelined. For scoreboards below, assume that an instruction waiting for a result from another function unit can pass through read operands at the same time the result is written. Also assume that an instruction in WB completing will allow a currently active instruction that is waiting on the same functional unit to issue in the same clock cycle in which the first instruction completes WB.

a. [20] <C.5> For this problem, use the RISC V pipeline of Section C.5 with the pipeline latencies from Figure C.29, but a fully pipelined FP unit, so the initiation interval is 1. Draw a timing diagram, similar to Figure C.32, showing the timing of each instruction's execution. How many clock cycles does each loop iteration take, counting from when the first instruction enters the WB stage to when the last instruction enters the WB stage?

b. [20] <C.8> Perform *static instruction reordering* to reorder the instructions to minimize the stalls for this loop, renaming registers where necessary. Use all the same assumptions as in (a). Draw a timing diagram, similar to Figure C.32, showing the timing of each instruction's execution. How many clock cycles does each loop iteration take, counting from when the first instruction enters the WB stage to when the last instruction enters the WB stage?

c. [20] <C.8> Using the original code above, consider how the instructions would have executed using scoreboarding, a form of dynamic scheduling. Draw a timing diagram, similar to Figure C.32, showing the timing of the instructions through stages IF, IS (issue), RO (read operands), EX (execution), and WR (write result). How many clock cycles does each loop iteration take, counting from when the first instruction enters the WB stage to when the last instruction enters the WB stage?

C.10 [25] <C.8> It is critical that the scoreboard be able to distinguish RAW and WAR hazards, because a WAR hazard requires stalling the instruction doing the writing until the instruction reading an operand initiates execution, but a RAW hazard requires delaying the reading instruction until the writing instruction finishes—just the opposite. For example, consider the sequence:

```
fmul.d   f0,f6,f4
fsub.d   f8,f0,f2
fadd.d   f2,f10,f2
```

The `fsub.d` depends on the `fmul.d` (a RAW hazard), thus the `fmul.d` must be allowed to complete before the `fsub.d`. If the `fmul.d` were stalled for the `fsub.d` due to the inability to distinguish between RAW and WAR hazards, the processor will deadlock. This sequence contains a WAR hazard between the `fadd.d` and the `fsub.d`, and the `fadd.d` cannot be allowed to complete until the `fsub.d` begins execution. The difficulty lies in distinguishing the RAW hazard between `fmul.d` and `fsub.d`, and the `WAR` hazard between the `fsub.d` and `fadd.d`. To see just why the three-instruction scenario is important, trace the handling of each instruction stage by stage through issue, read operands, execute, and write result. Assume that each scoreboard stage other than execute takes 1 clock cycle. Assume that the `fmul.d` instruction requires 3 clock cycles to execute and that the `fsub.d` and `fadd.d` instructions each take 1 cycle to execute. Finally, assume that the processor has two multiply function units and two add function units. Present the trace as follows.

1. Make a table with the column headings Instruction, Issue, Read Operands, Execute, Write Result, and Comment. In the first column, list the instructions in program order (be generous with space between instructions; larger table cells will better hold the results of your analysis). Start the table by writing a 1 in the Issue column of the `fmul.d` instruction row to show that `fmul.d` completes the issue stage in clock cycle 1. Now, fill in the stage columns of the table through the cycle at which the scoreboard first stalls an instruction.
2. For a stalled instruction write the words "waiting at clock cycle X," where X is the number of the current clock cycle, in the appropriate table column to show that the scoreboard is resolving an RAW or WAR hazard by stalling that stage. In the Comment column, state what type of hazard and what dependent instruction is causing the wait.

3. Adding the words "completes with clock cycle Y" to a "waiting" table entry, fill in the rest of the table through the time when all instructions are complete. For an instruction that stalled, add a description in the Comments column telling why the wait ended when it did and how deadlock was avoided (Hint: Think about how WAW hazards are prevented and what this implies about active instruction sequences.). Note the completion order of the three instructions as compared to their program order.

C.11 [10/10/10] < C.5 > For this problem, you will create a series of small snippets that illustrate the issues that arise when using functional units with different latencies. For each one, draw a timing diagram similar to Figure C.32 that illustrates each concept, and clearly indicate the problem.

a. [10] < C.5 > Demonstrate, using code different from that used in Figure C.32, the structural hazard of having the hardware for only one MEM and WB stage.

b. [10] < C.5 > Demonstrate a WAW hazard requiring a stall.

D.1 Introduction D-2
D.2 Advanced Topics in Disk Storage D-2
D.3 Definition and Examples of Real Faults and Failures D-10
D.4 I/O Performance, Reliability Measures, and Benchmarks D-15
D.5 A Little Queuing Theory D-23
D.6 Crosscutting Issues D-34
D.7 Designing and Evaluating an I/O System—The Internet Archive Cluster D-36
D.8 Putting It All Together: NetApp FAS6000 Filer D-41
D.9 Fallacies and Pitfalls D-43
D.10 Concluding Remarks D-47
D.11 Historical Perspective and References D-48
Case Studies with Exercises by Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau D-48

D

Storage Systems

I think Silicon Valley was misnamed. If you look back at the dollars shipped in products in the last decade, there has been more revenue from magnetic disks than from silicon. They ought to rename the place Iron Oxide Valley.

Al Hoagland
A pioneer of magnetic disks (1982)

Combining bandwidth and storage … enables swift and reliable access to the ever expanding troves of content on the proliferating disks and … repositories of the Internet … the capacity of storage arrays of all kinds is rocketing ahead of the advance of computer performance.

George Gilder
"The End Is Drawing Nigh,"
Forbes ASAP (April 4, 2000)

D.1 Introduction

The popularity of Internet services such as search engines and auctions has enhanced the importance of I/O for computers, since no one would want a desktop computer that couldn't access the Internet. This rise in importance of I/O is reflected by the names of our times. The 1960s to 1980s were called the Computing Revolution; the period since 1990 has been called the Information Age, with concerns focused on advances in information technology versus raw computational power. Internet services depend upon massive storage, which is the focus of this chapter, and networking, which is the focus of Appendix F.

This shift in focus from computation to communication and storage of information emphasizes reliability and scalability as well as cost-performance. Although it is frustrating when a program crashes, people become hysterical if they lose their data; hence, storage systems are typically held to a higher standard of dependability than the rest of the computer. Dependability is the bedrock of storage, yet it also has its own rich performance theory—queuing theory—that balances throughput versus response time. The software that determines which processor features get used is the compiler, but the operating system usurps that role for storage.

Thus, storage has a different, multifaceted culture from processors, yet it is still found within the architecture tent. We start our exploration with advances in magnetic disks, as they are the dominant storage device today in desktop and server computers. We assume that readers are already familiar with the basics of storage devices, some of which were covered in Chapter 1.

D.2 Advanced Topics in Disk Storage

The disk industry historically has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as improvement in *areal density*, measured in bits per square inch:

$$\text{Areal density} = \frac{\text{Tracks}}{\text{Inch}} \text{on a disk surface} \times \frac{\text{Bits}}{\text{Inch}} \text{on a track}$$

Through about 1988, the rate of improvement of areal density was 29% per year, thus doubling density every 3 years. Between then and about 1996, the rate improved to 60% per year, quadrupling density every 3 years and matching the traditional rate of DRAMs. From 1997 to about 2003, the rate increased to 100%, doubling every year. After the innovations that allowed this renaissances had largely played out, the rate has dropped recently to about 30% per year. In 2011, the highest density in commercial products is 400 billion bits per square inch. Cost per gigabyte has dropped at least as fast as areal density has increased, with smaller diameter drives playing the larger role in this improvement. Costs per gigabyte improved by almost a factor of 1,000,000 between 1983 and 2011.

Magnetic disks have been challenged many times for supremacy of secondary storage. Figure D.1 shows one reason: the fabled *access time gap* between disks and DRAM. DRAM latency is about 100,000 times less than disk, and that performance advantage costs 30 to 150 times more per gigabyte for DRAM.

The bandwidth gap is more complex. For example, a fast disk in 2011 transfers at 200 MB/sec from the disk media with 600 GB of storage and costs about $400. A 4 GB DRAM module costing about $200 in 2011 could transfer at 16,000 MB/sec (see Chapter 2), giving the DRAM module about 80 times higher bandwidth than the disk. However, the bandwidth per GB is 6000 times higher for DRAM, and the bandwidth per dollar is 160 times higher.

Many have tried to invent a technology cheaper than DRAM but faster than disk to fill that gap, but thus far all have failed. Challengers have never had a product to market at the right time. By the time a new product ships, DRAMs and disks have made advances as predicted earlier, costs have dropped accordingly, and the challenging product is immediately obsolete.

The closest challenger is Flash memory. This semiconductor memory is nonvolatile like disks, and it has about the same bandwidth as disks, but latency is 100 to 1000 times faster than disk. In 2011, the price per gigabyte of Flash was 15 to 20 times cheaper than DRAM. Flash is popular in cell phones because it comes in much smaller capacities and it is more power efficient than disks, despite the cost per gigabyte being 15 to 25 times higher than disks. Unlike disks and DRAM,

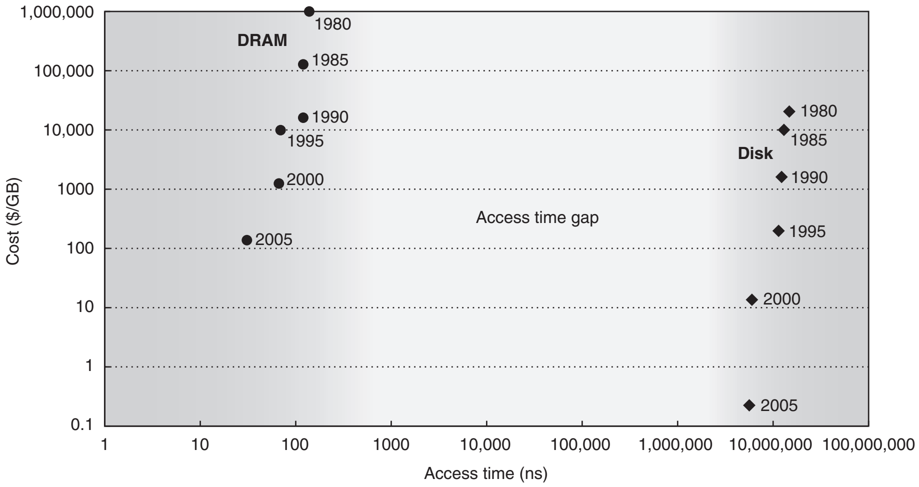


Figure D.1 Cost versus access time for DRAM and magnetic disk in 1980, 1985, 1990, 1995, 2000, and 2005. The two-order-of-magnitude gap in cost and five-order-of-magnitude gap in access times between semiconductor memory and rotating magnetic disks have inspired a host of competing technologies to try to fill them. So far, such attempts have been made obsolete before production by improvements in magnetic disks, DRAMs, or both. Note that between 1990 and 2005 the cost per gigabyte DRAM chips made less improvement, while disk cost made dramatic improvement.

Flash memory bits wear out—typically limited to 1 million writes—and so they are not popular in desktop and server computers.

While disks will remain viable for the foreseeable future, the conventional sector-track-cylinder model did not. The assumptions of the model are that nearby blocks are on the same track, blocks in the same cylinder take less time to access since there is no seek time, and some tracks are closer than others.

First, disks started offering higher-level intelligent interfaces, like ATA and SCSI, when they included a microprocessor inside a disk. To speed up sequential transfers, these higher-level interfaces organize disks more like tapes than like random access devices. The logical blocks are ordered in serpentine fashion across a single surface, trying to capture all the sectors that are recorded at the same bit density. (Disks vary the recording density since it is hard for the electronics to keep up with the blocks spinning much faster on the outer tracks, and lowering linear density simplifies the task.) Hence, sequential blocks may be on different tracks. We will see later in Figure D.22 on page D-45 an illustration of the fallacy of assuming the conventional sector-track model when working with modern disks.

Second, shortly after the microprocessors appeared inside disks, the disks included buffers to hold the data until the computer was ready to accept it, and later caches to avoid read accesses. They were joined by a command queue that allowed the disk to decide in what order to perform the commands to maximize performance while maintaining correct behavior. Figure D.2 shows how a queue depth of 50 can double the number of I/Os per second of random I/Os due to better scheduling of accesses. Although it's unlikely that a system would really have 256 commands in a queue, it would triple the number of I/Os per second. Given buffers, caches, and out-of-order accesses, an accurate performance model of a real disk is much more complicated than sector-track-cylinder.

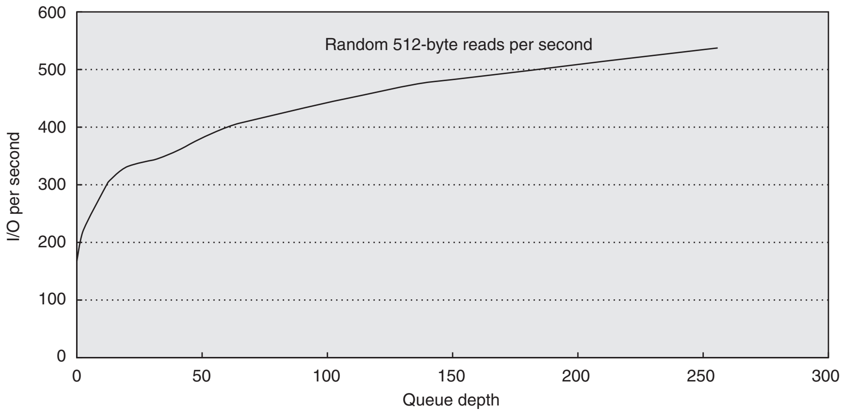


Figure D.2 Throughput versus command queue depth using random 512-byte reads. The disk performs 170 reads per second starting at no command queue and doubles performance at 50 and triples at 256 [Anderson 2003].

Finally, the number of platters shrank from 12 in the past to 4 or even 1 today, so the cylinder has less importance than before because the percentage of data in a cylinder is much less.

Disk Power

Power is an increasing concern for disks as well as for processors. A typical ATA disk in 2011 might use 9 watts when idle, 11 watts when reading or writing, and 13 watts when seeking. Because it is more efficient to spin smaller mass, smaller-diameter disks can save power. One formula that indicates the importance of rotation speed and the size of the platters for the power consumed by the disk motor is the following [Gurumurthi et al. 2005]:

$$\text{Power} \approx \text{Diameter}^{4.6} \times \text{RPM}^{2.8} \times \text{Number of platters}$$

Thus, smaller platters, slower rotation, and fewer platters all help reduce disk motor power, and most of the power is in the motor.

Figure D.3 shows the specifications of two 3.5-inch disks in 2011. The *Serial ATA* (SATA) disks shoot for high capacity and the best cost per gigabyte, so the 2000 GB drives cost less than \$0.05 per gigabyte. They use the widest platters that fit the form factor and use four or five of them, but they spin at 5900 RPM and seek relatively slowly to allow a higher areal density and to lower power. The corresponding *Serial Attach SCSI* (SAS) drive aims at performance, so it spins at 15,000 RPM and seeks much faster. It uses a lower areal density to spin at that high rate. To reduce power, the platter is much narrower than the form factor. This combination reduces capacity of the SAS drive to 600 GB.

The cost per gigabyte is about a factor of five better for the SATA drives, and, conversely, the cost per I/O per second or MB transferred per second is about a factor of five better for the SAS drives. Despite using smaller platters and many fewer of them, the SAS disks use twice the power of the SATA drives, due to the much faster RPM and seeks.

	Capacity (GB)	Price	Platters	RPM	Diameter (inches)	Average seek (ms)	Power (watts)	I/O/sec	Disk BW (MB/sec)	Buffer BW (MB/sec)	Buffer size (MB)	MTTF (hrs)
SATA	2000	\$85	4	5900	3.7	16	12	47	45–95	300	32	0.6 M
SAS	600	\$400	4	15,000	2.6	3–4	16	285	122–204	750	16	1.6 M

Figure D.3 Serial ATA (SATA) versus Serial Attach SCSI (SAS) drives in 3.5-inch form factor in 2011. The I/Os per second were calculated using the average seek plus the time for one-half rotation plus the time to transfer one sector of 512 KB.

Advanced Topics in Disk Arrays

An innovation that improves both dependability and performance of storage systems is *disk arrays*. One argument for arrays is that potential throughput can be increased by having many disk drives and, hence, many disk arms, rather than fewer large drives. Simply spreading data over multiple disks, called *striping*, automatically forces accesses to several disks if the data files are large. (Although arrays improve throughput, latency is not necessarily improved.) As we saw in Chapter 1, the drawback is that with more devices, dependability decreases: N devices generally have $1/N$ the reliability of a single device.

Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability is improved by adding redundant disks to the array to tolerate faults. That is, if a single disk fails, the lost information is reconstructed from redundant information. The only danger is in having another disk fail during the *mean time to repair* (MTTR). Since the *mean time to failure* (MTTF) of disks is tens of years, and the MTTR is measured in hours, redundancy can make the measured reliability of many disks much higher than that of a single disk.

Such redundant disk arrays have become known by the acronym *RAID*, which originally stood for *redundant array of inexpensive disks*, although some prefer the word *independent* for *I* in the acronym. The ability to recover from failures plus the higher throughput, measured as either megabytes per second or I/Os per second, make RAID attractive. When combined with the advantages of smaller size and lower power of small-diameter drives, RAIDs now dominate large-scale storage systems.

Figure D.4 summarizes the five standard RAID levels, showing how eight disks of user data must be supplemented by redundant or check disks at each RAID level, and it lists the pros and cons of each level. The standard RAID levels are well documented, so we will just do a quick review here and discuss advanced levels in more depth.

- *RAID 0*—It has no redundancy and is sometimes nicknamed *JBOD*, for *just a bunch of disks*, although the data may be striped across the disks in the array. This level is generally included to act as a measuring stick for the other RAID levels in terms of cost, performance, and dependability.
- *RAID 1*—Also called *mirroring* or *shadowing*, there are two copies of every piece of data. It is the simplest and oldest disk redundancy scheme, but it also has the highest cost. Some array controllers will optimize read performance by allowing the mirrored disks to act independently for reads, but this optimization means it may take longer for the mirrored writes to complete.
- *RAID 2*—This organization was inspired by applying memory-style error-correcting codes (ECCs) to disks. It was included because there was such a disk array product at the time of the original RAID paper, but none since then as other RAID organizations are more attractive.
- *RAID 3*—Since the higher-level disk interfaces understand the health of a disk, it's easy to figure out which disk failed. Designers realized that if one extra disk

RAID level		Disk failures tolerated, check space overhead for 8 data disks	Pros	Cons	Company products
0	Nonredundant striped	0 failures, 0 check disks	No space overhead	No protection	Widely used
1	Mirrored	1 failure, 8 check disks	No parity calculation; fast recovery; small writes faster than higher RAIDs; fast reads	Highest check storage overhead	EMC, HP (Tandem), IBM
2	Memory-style ECC	1 failure, 4 check disks	Doesn't rely on failed disk to self-diagnose	~ Log 2 check storage overhead	Not used
3	Bit-interleaved parity	1 failure, 1 check disk	Low check overhead; high bandwidth for large reads or writes	No support for small, random reads or writes	Storage Concepts
4	Block-interleaved parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads	Parity disk is small write bottleneck	Network Appliance
5	Block-interleaved distributed parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads and writes	Small writes → 4 disk accesses	Widely used
6	Row-diagonal parity, EVEN-ODD	2 failures, 2 check disks	Protects against 2 disk failures	Small writes → 6 disk accesses; 2 × check overhead	Network Appliance

Figure D.4 RAID levels, their fault tolerance, and their overhead in redundant disks. The paper that introduced the term *RAID* [Patterson, Gibson, and Katz 1987] used a numerical classification that has become popular. In fact, the nonredundant disk array is often called *RAID 0*, indicating that the data are striped across several disks but without redundancy. Note that mirroring (RAID 1) in this instance can survive up to eight disk failures provided only one disk of each mirrored pair fails; worst case is both disks in a mirrored pair fail. In 2011, there may be no commercial implementations of RAID 2; the rest are found in a wide range of products. RAID 0+1, 1+0, 01, 10, and 6 are discussed in the text.

contains the parity of the information in the data disks, a single disk allows recovery from a disk failure. The data are organized in stripes, with *N* data blocks and one parity block. When a failure occurs, we just "subtract" the good data from the good blocks, and what remains is the missing data. (This works whether the failed disk is a data disk or the parity disk.) RAID 3 assumes that the data are spread across all disks on reads and writes, which is attractive when reading or writing large amounts of data.

- *RAID 4*—Many applications are dominated by small accesses. Since sectors have their own error checking, you can safely increase the number of reads per second by allowing each disk to perform independent reads. It would seem that writes would still be slow, if you have to read every disk to calculate parity. To increase the number of writes per second, an alternative approach involves only two disks. First, the array reads the old data that are about to be overwritten, and then calculates what bits would change before it writes the new data. It then reads the old value of the parity on the check disks, updates parity according to the list of changes, and then writes the new value of parity to the check disk. Hence, these so-called "small writes" are still slower than small reads—they involve four disks accesses—but they are faster than if you had to read all disks on every write. RAID 4 has the same low check disk overhead as RAID 3, and it can still do large reads and writes as fast as RAID 3 in addition to small reads and writes, but control is more complex.

- *RAID 5*—Note that a performance flaw for small writes in RAID 4 is that they all must read and write the same check disk, so it is a performance bottleneck. RAID 5 simply distributes the parity information across all disks in the array, thereby removing the bottleneck. The parity block in each stripe is rotated so that parity is spread evenly across all disks. The disk array controller must now calculate which disk has the parity for when it wants to write a given block, but that can be a simple calculation. RAID 5 has the same low check disk overhead as RAID 3 and 4, and it can do the large reads and writes of RAID 3 and the small reads of RAID 4, but it has higher small write bandwidth than RAID 4. Nevertheless, RAID 5 requires the most sophisticated controller of the classic RAID levels.

Having completed our quick review of the classic RAID levels, we can now look at two levels that have become popular since RAID was introduced.

RAID 10 versus 01 (or 1+0 versus RAID 0+1)

One topic not always described in the RAID literature involves how mirroring in RAID 1 interacts with striping. Suppose you had, say, four disks' worth of data to store and eight physical disks to use. Would you create four pairs of disks—each organized as RAID 1—and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks—each organized as RAID 0—and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1+0 or RAID 10 ("striped mirrors") and the latter RAID 0+1 or RAID 01 ("mirrored stripes").

RAID 6: Beyond a Single Disk Failure

The parity-based schemes of the RAID 1 to 5 protect against a single self-identifying failure; however, if an operator accidentally replaces the wrong disk during a failure, then the disk array will experience two failures, and data will be lost. Another concern is that since disk bandwidth is growing more slowly than disk capacity, the MTTR of a disk in a RAID system is increasing, which in turn increases the chances of a second failure. For example, a 500 GB SATA disk could take about 3 hours to read sequentially assuming no interference. Given that the damaged RAID is likely to continue to serve data, reconstruction could be stretched considerably, thereby increasing MTTR. Besides increasing reconstruction time, another concern is that reading much more data during reconstruction means increasing the chance of an uncorrectable media failure, which would result in data loss. Other arguments for concern about simultaneous multiple failures are the increasing number of disks in arrays and the use of ATA disks, which are slower and larger than SCSI disks.

Hence, over the years, there has been growing interest in protecting against more than one failure. Network Appliance (NetApp), for example, started by building RAID 4 file servers. As double failures were becoming a danger to customers, they created a more robust scheme to protect data, called *row-diagonal parity* or *RAID-DP* [Corbett et al. 2004]. Like the standard RAID schemes, row-diagonal parity uses redundant space based on a parity calculation on a per-stripe basis. Since it is protecting against a double failure, it adds two check blocks per stripe of data. Let's assume there are $p+1$ disks total, so $p-1$ disks have data. Figure D.5 shows the case when p is 5.

The row parity disk is just like in RAID 4; it contains the even parity across the other four data blocks in its stripe. Each block of the diagonal parity disk contains the even parity of the blocks in the same diagonal. Note that each diagonal does not cover one disk; for example, diagonal 0 does not cover disk 1. Hence, we need just $p-1$ diagonals to protect the p disks, so the disk only has diagonals 0 to 3 in Figure D.5.

Let's see how row-diagonal parity works by assuming that data disks 1 and 3 fail in Figure D.5. We can't perform the standard RAID recovery using the first row using row parity, since it is missing two data blocks from disks 1 and 3. However, we can perform recovery on diagonal 0, since it is only missing the data block associated with disk 3. Thus, row-diagonal parity starts by recovering one of the four blocks on the failed disk in this example using diagonal parity. Since each diagonal misses one disk, and all diagonals miss a different disk, two diagonals are only missing one block. They are diagonals 0 and 2 in this example, so we next restore the block from diagonal 2 from failed disk 1. When the data for those blocks have been recovered, then the standard RAID recovery scheme can be used to

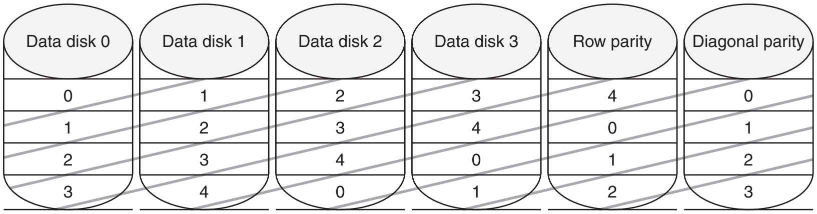


Figure D.5 Row diagonal parity for $p = 5$, which protects four data disks from double failures [Corbett et al. 2004]. This figure shows the diagonal groups for which parity is calculated and stored in the diagonal parity disk. Although this shows all the check data in separate disks for row parity and diagonal parity as in RAID 4, there is a rotated version of row-diagonal parity that is analogous to RAID 5. Parameter p must be prime and greater than 2; however, you can make p larger than the number of data disks by assuming that the missing disks have all zeros and the scheme still works. This trick makes it easy to add disks to an existing system. NetApp picks p to be 257, which allows the system to grow to up to 256 data disks.

recover two more blocks in the standard RAID 4 stripes 0 and 2, which in turn allows us to recover more diagonals. This process continues until two failed disks are completely restored.

The EVEN-ODD scheme developed earlier by researchers at IBM is similar to row diagonal parity, but it has a bit more computation during operation and recovery [Blaum 1995]. Papers that are more recent show how to expand EVEN-ODD to protect against three failures [Blaum, Bruck, and Vardy 1996; Blaum et al. 2001].

D.3 Definition and Examples of Real Faults and Failures

Although people may be willing to live with a computer that occasionally crashes and forces all programs to be restarted, they insist that their information is never lost. The prime directive for storage is then to remember information, no matter what happens.

Chapter 1 covered the basics of dependability, and this section expands that information to give the standard definitions and examples of failures.

The first step is to clarify confusion over terms. The terms *fault*, *error*, and *failure* are often used interchangeably, but they have different meanings in the dependability literature. For example, is a programming mistake a fault, error, or failure? Does it matter whether we are talking about when it was designed or when the program is run? If the running program doesn't exercise the mistake, is it still a fault/error/failure? Try another one. Suppose an alpha particle hits a DRAM memory cell. Is it a fault/error/failure if it doesn't change the value? Is it a fault/error/failure if the memory doesn't access the changed bit? Did a fault/error/failure still occur if the memory had error correction and delivered the corrected value to the CPU? You get the drift of the difficulties. Clearly, we need precise definitions to discuss such events intelligently.

To avoid such imprecision, this subsection is based on the terminology used by Laprie [1985] and Gray and Siewiorek [1991], endorsed by IFIP Working Group 10.4 and the IEEE Computer Society Technical Committee on Fault Tolerance. We talk about a system as a single module, but the terminology applies to submodules recursively. Let's start with a definition of *dependability*:

> *Computer system* dependability *is the quality of delivered service such that reliance can justifiably be placed on this service. The* service *delivered by a system is its observed* actual behavior *as perceived by other system(s) interacting with this system's users. Each module also has an ideal* specified behavior, *where a* service specification *is an agreed description of the expected behavior. A system* failure *occurs when the actual behavior deviates from the specified behavior. The failure occurred because of an* error, *a defect in that module. The cause of an error is a* fault.
>
> *When a fault occurs, it creates a* latent error, *which becomes* effective *when it is activated; when the error actually affects the delivered service, a failure occurs. The*

> *time between the occurrence of an error and the resulting failure is the* error latency. *Thus, an error is the manifestation* in the system *of a fault, and a failure is the manifestation* on the service *of an error. [p. 3]*

Let's go back to our motivating examples above. A programming mistake is a *fault*. The consequence is an *error* (or *latent error*) in the software. Upon activation, the error becomes *effective*. When this effective error produces erroneous data that affect the delivered service, a *failure* occurs.

An alpha particle hitting a DRAM can be considered a fault. If it changes the memory, it creates an error. The error will remain latent until the affected memory word is read. If the effective word error affects the delivered service, a failure occurs. If ECC corrected the error, a failure would not occur.

A mistake by a human operator is a fault. The resulting altered data is an error. It is latent until activated, and so on as before.

To clarify, the relationship among faults, errors, and failures is as follows:

- A fault creates one or more latent errors.
- The properties of errors are (1) a latent error becomes effective once activated; (2) an error may cycle between its latent and effective states; and (3) an effective error often propagates from one component to another, thereby creating new errors. Thus, either an effective error is a formerly latent error in that component or it has propagated from another error in that component or from elsewhere.
- A component failure occurs when the error affects the delivered service.
- These properties are recursive and apply to any component in the system.

Gray and Siewiorek classified faults into four categories according to their cause:

1. *Hardware faults*—Devices that fail, such as perhaps due to an alpha particle hitting a memory cell
2. *Design faults*—Faults in software (usually) and hardware design (occasionally)
3. *Operation faults*—Mistakes by operations and maintenance personnel
4. *Environmental faults*—Fire, flood, earthquake, power failure, and sabotage

Faults are also classified by their duration into transient, intermittent, and permanent [Nelson 1990]. *Transient faults* exist for a limited time and are not recurring. *Intermittent faults* cause a system to oscillate between faulty and fault-free operation. *Permanent faults* do not correct themselves with the passing of time.

Now that we have defined the difference between faults, errors, and failures, we are ready to see some real-world examples. Publications of real error rates are rare for two reasons. First, academics rarely have access to significant hardware resources to measure. Second, industrial researchers are rarely allowed to publish failure information for fear that it would be used against their companies in the marketplace. A few exceptions follow.

Berkeley's Tertiary Disk

The Tertiary Disk project at the University of California created an art image server for the Fine Arts Museums of San Francisco in 2000. This database consisted of high-quality images of over 70,000 artworks [Talagala et al., 2000]. The database was stored on a cluster, which consisted of 20 PCs connected by a switched Ethernet and containing 368 disks. It occupied seven 7-foot-high racks.

Figure D.6 shows the failure rates of the various components of Tertiary Disk. In advance of building the system, the designers assumed that SCSI data disks would be the least reliable part of the system, as they are both mechanical and plentiful. Next would be the IDE disks since there were fewer of them, then the power supplies, followed by integrated circuits. They assumed that passive devices such as cables would scarcely ever fail.

Figure D.6 shatters some of those assumptions. Since the designers followed the manufacturer's advice of making sure the disk enclosures had reduced vibration and good cooling, the data disks were very reliable. In contrast, the PC chassis containing the IDE/ATA disks did not afford the same environmental controls. (The IDE/ATA disks did not store data but helped the application and operating

Component	Total in system	Total failed	Percentage failed
SCSI controller	44	1	2.3%
SCSI cable	39	1	2.6%
SCSI disk	368	7	1.9%
IDE/ATA disk	24	6	25.0%
Disk enclosure—backplane	46	13	28.3%
Disk enclosure—power supply	92	3	3.3%
Ethernet controller	20	1	5.0%
Ethernet switch	2	1	50.0%
Ethernet cable	42	1	2.3%
CPU/motherboard	20	0	0%

Figure D.6 Failures of components in Tertiary Disk over 18 months of operation. For each type of component, the table shows the total number in the system, the number that failed, and the percentage failure rate. Disk enclosures have two entries in the table because they had two types of problems: backplane integrity failures and power supply failures. Since each enclosure had two power supplies, a power supply failure did not affect availability. This cluster of 20 PCs, contained in seven 7-foot-high, 19-inch-wide racks, hosted 368 8.4 GB, 7200 RPM, 3.5-inch IBM disks. The PCs were P6-200 MHz with 96 MB of DRAM each. They ran FreeBSD 3.0, and the hosts were connected via switched 100 Mbit/sec Ethernet. All SCSI disks were connected to two PCs via double-ended SCSI chains to support RAID 1. The primary application was called the Zoom Project, which in 1998 was the world's largest art image database, with 72,000 images. See Talagala et al. [2000b].

system to boot the PCs.) Figure D.6 shows that the SCSI backplane, cables, and Ethernet cables were no more reliable than the data disks themselves!

As Tertiary Disk was a large system with many redundant components, it could survive this wide range of failures. Components were connected and mirrored images were placed so that no single failure could make any image unavailable. This strategy, which initially appeared to be overkill, proved to be vital.

This experience also demonstrated the difference between transient faults and hard faults. Virtually all the failures in Figure D.6 appeared first as transient faults. It was up to the operator to decide if the behavior was so poor that they needed to be replaced or if they could continue. In fact, the word "failure" was not used; instead, the group borrowed terms normally used for dealing with problem employees, with the operator deciding whether a problem component should or should not be "fired."

Tandem

The next example comes from industry. Gray [1990] collected data on faults for Tandem Computers, which was one of the pioneering companies in fault-tolerant computing and used primarily for databases. Figure D.7 graphs the faults that caused system failures between 1985 and 1989 in absolute faults per system and in percentage of faults encountered. The data show a clear improvement in the reliability of hardware and maintenance. Disks in 1985 required yearly service by Tandem, but they were replaced by disks that required no scheduled maintenance. Shrinking numbers of chips and connectors per system plus software's ability to tolerate hardware faults reduced hardware's contribution to only 7% of failures by 1989. Moreover, when hardware was at fault, software embedded in the hardware device (firmware) was often the culprit. The data indicate that software in 1989 was the major source of reported outages (62%), followed by system operations (15%).

The problem with any such statistics is that the data only refer to what is reported; for example, environmental failures due to power outages were not reported to Tandem because they were seen as a local problem. Data on operation faults are very difficult to collect because operators must report personal mistakes, which may affect the opinion of their managers, which in turn can affect job security and pay raises. Gray suggested that both environmental faults and operator faults are underreported. His study concluded that achieving higher availability requires improvement in software quality and software fault tolerance, simpler operations, and tolerance of operational faults.

Other Studies of the Role of Operators in Dependability

While Tertiary Disk and Tandem are storage-oriented dependability studies, we need to look outside storage to find better measurements on the role of humans in failures. Murphy and Gent [1995] tried to improve the accuracy of data on

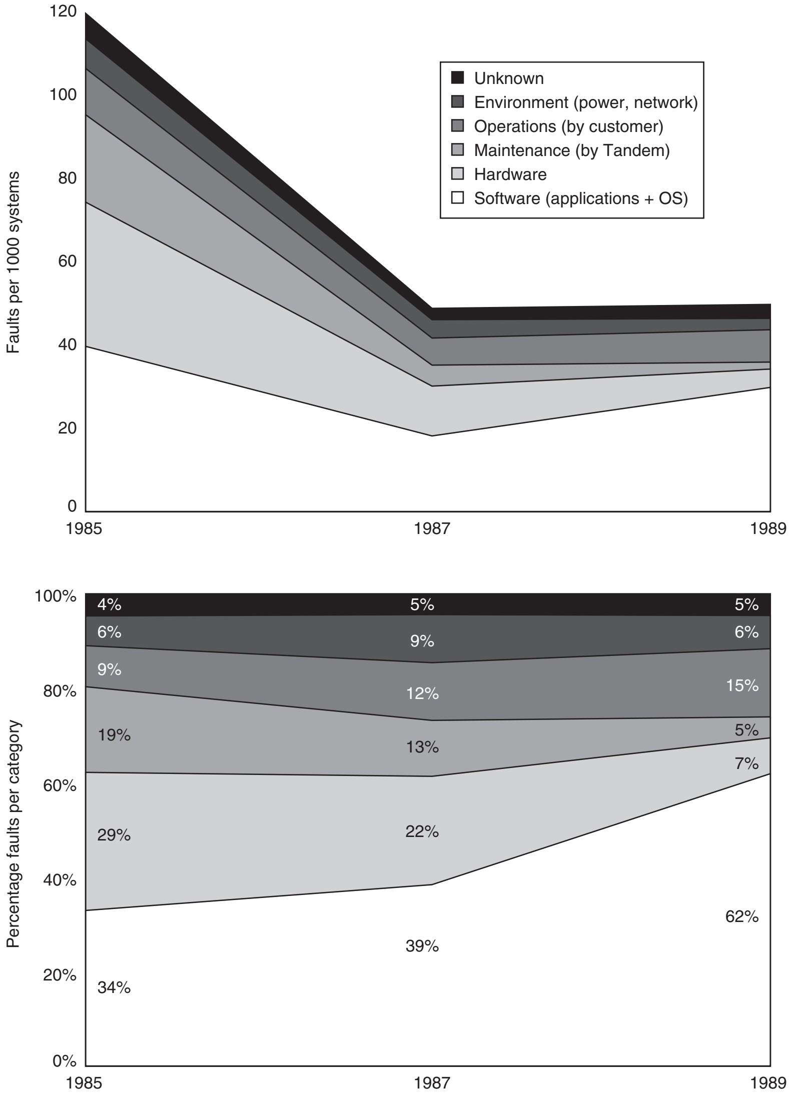


Figure D.7 Faults in Tandem between 1985 and 1989. Gray [1990] collected these data for fault-tolerant Tandem Computers based on reports of component failures by customers.

operator faults by having the system automatically prompt the operator on each boot for the reason for that reboot. They classified consecutive crashes to the same fault as operator fault and included operator actions that directly resulted in crashes, such as giving parameters bad values, bad configurations, and bad application installation. Although they believed that operator error is under-reported, they did get more accurate information than did Gray, who relied on a form that the operator filled out and then sent up the management chain. The hardware/operating system went from causing 70% of the failures in VAX systems in 1985 to 28% in 1993, and failures due to operators rose from 15% to 52% in that same period. Murphy and Gent expected managing systems to be the primary dependability challenge in the future.

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts 30 minutes. These detailed disruption reports do not suffer from the self-reporting problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994, and Enriquez [2001] did a follow-up study for the first half of 2001. Although there was a significant improvement in failures due to overloading of the network over the years, failures due to humans increased, from about one-third to two-thirds of the customer-outage minutes.

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as memory ECC and RAID. At least some operating systems are considering reliability implications before adding new features, so in 2011 the failures largely occurred elsewhere.

Although failures may be initiated due to faults by operators, it is a poor reflection on the state of the art of systems that the processes of maintenance and upgrading are so error prone. Most storage vendors claim today that customers spend much more on managing storage over its lifetime than they do on purchasing the storage. Thus, the challenge for dependable storage systems of the future is either to tolerate faults by operators or to avoid faults by simplifying the tasks of system administration. Note that RAID 6 allows the storage system to survive even if the operator mistakenly replaces a good disk.

We have now covered the bedrock issue of dependability, giving definitions, case studies, and techniques to improve it. The next step in the storage tour is performance.

D.4 I/O Performance, Reliability Measures, and Benchmarks

I/O performance has measures that have no counterparts in design. One of these is diversity: Which I/O devices can connect to the computer system? Another is capacity: How many I/O devices can connect to a computer system?

In addition to these unique measures, the traditional measures of performance (namely, response time and throughput) also apply to I/O. (I/O throughput is sometimes called *I/O bandwidth* and response time is sometimes called *latency.*) The next two figures offer insight into how response time and throughput trade off

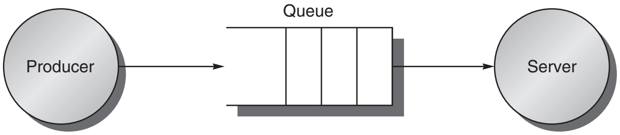


Figure D.8 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

against each other. Figure D.8 shows the simple producer-server model. The producer creates tasks to be performed and places them in a buffer; the server takes tasks from the first in, first out buffer and performs them.

Response time is defined as the time a task takes from the moment it is placed in the buffer until the server finishes the task. Throughput is simply the average number of tasks completed by the server over a time period. To get the highest possible throughput, the server should never be idle, thus the buffer should never be empty. Response time, on the other hand, counts time spent in the buffer, so an empty buffer shrinks it.

Another measure of I/O performance is the interference of I/O with processor execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how much longer a process will take because of I/O for another process.

Throughput versus Response Time

Figure D.9 shows throughput versus response time (or latency) for a typical I/O system. The knee of the curve is the area where a little more throughput results in much longer response time or, conversely, a little shorter response time results in much lower throughput.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure D.10 suggests an answer. An interaction, or *transaction*, with a computer is divided into three parts:

1. *Entry time*—The time for the user to enter the command.
2. *System response time*—The time between when the user enters the command and the complete response is displayed.
3. *Think time*—The time from the reception of the response until the user begins to enter the next command.

The sum of these three parts is called the *transaction time*. Several studies report that user productivity is inversely proportional to transaction time. The results in Figure D.10 show that cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from

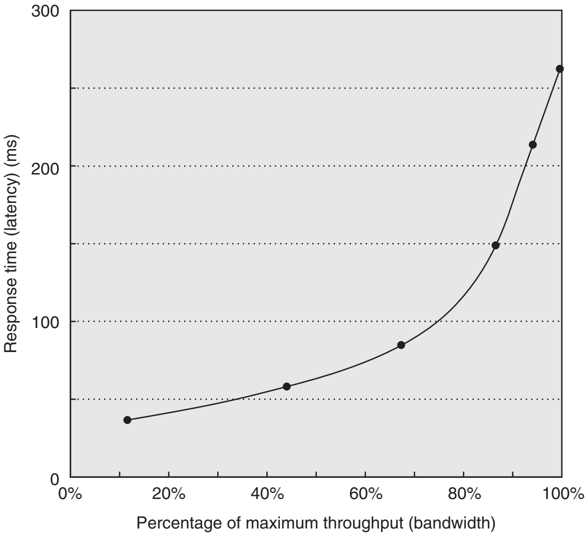


Figure D.9 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note also that the independent variable in this curve is implicit; to trace the curve, you typically vary load (concurrency). Chen et al. [1990] collected these data for an array of magnetic disks.

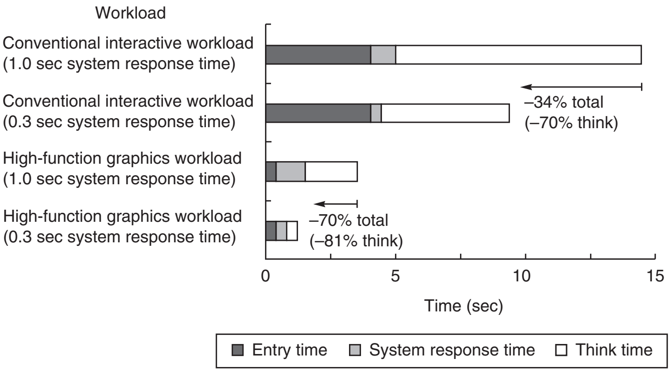


Figure D.10 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. Reduction in response time actually decreases transaction time by more than just the response time reduction. (From Brady [1986].)

I/O benchmark	Response time restriction	Throughput metric
TPC-C: Complex Query OLTP	≥90% of transaction must meet response time limit; 5 seconds for most types of transactions	New order transactions per minute
TPC-W: Transactional Web benchmark	≥90% of Web interactions must meet response time limit; 3 seconds for most types of Web interactions	Web interactions per second
SPECsfs97	Average response time ≤40 ms	NFS operations per second

Figure D.11 Response time restrictions for three I/O benchmarks.

the graphics transaction. This implausible result is explained by human nature: People need less time to think when given a faster response. Although this study is 20 years old, response times are often still much slower than 1 second, even if processors are 1000 times faster. Examples of long delays include starting an application on a desktop PC due to many disk I/Os, or network delays when clicking on Web links.

To reflect the importance of response time to user productivity, I/O benchmarks also address the response time versus throughput trade-off. Figure D.11 shows the response time bounds for three I/O benchmarks. They report maximum throughput given either that 90% of response times must be less than a limit or that the average response time must be less than a limit.

Let's next look at these benchmarks in more detail.

Transaction-Processing Benchmarks

Transaction processing (TP, or OLTP for online transaction processing) is chiefly concerned with *I/O rate* (the number of disk accesses per second), as opposed to *data rate* (measured as bytes of data per second). TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. Suppose, for example, that a bank's computer fails when a customer tries to withdraw money from an ATM. The TP system would guarantee that the account is debited if the customer received the money *and* that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

As mentioned in Chapter 1, two dozen members of the TP community conspired to form a benchmark for the industry and, to avoid the wrath of their legal departments, published the report anonymously [Anon. et al. 1985]. This report led to the *Transaction Processing Council*, which in turn has led to eight benchmarks since its founding. Figure D.12 summarizes these benchmarks.

Let's describe TPC-C to give a flavor of these benchmarks. TPC-C uses a database to simulate an order-entry environment of a wholesale supplier, including

Benchmark	**Data size (GB)**	**Performance metric**	**Date of first results**
A: debit credit (retired)	0.1–10	Transactions per second	July 1990
B: batch debit credit (retired)	0.1–10	Transactions per second	July 1991
C: complex query OLTP	100–3000 (minimum 0.07 * TPM)	New order transactions per minute (TPM)	September 1992
D: decision support (retired)	100, 300, 1000	Queries per hour	December 1995
H: ad hoc decision support	100, 300, 1000	Queries per hour	October 1999
R: business reporting decision support (retired)	1000	Queries per hour	August 1999
W: transactional Web benchmark	≈50, 500	Web interactions per second	July 2000
App: application server and Web services benchmark	≈2500	Web service interactions per second (SIPS)	June 2005

Figure D.12 Transaction Processing Council benchmarks. The summary results include both the performance metric and the price-performance of that metric. TPC-A, TPC-B, TPC-D, and TPC-R were retired.

entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. It runs five concurrent transactions of varying complexity, and the database includes nine tables with a scalable range of records and customers. TPC-C is measured in transactions per minute (tpmC) and in price of system, including hardware, software, and three years of maintenance support. Figure 1.17 on page 42 in Chapter 1 describes the top systems in performance and cost-performance for TPC-C.

These TPC benchmarks were the first—and in some cases still the only ones—that have these unusual characteristics:

- *Price is included with the benchmark results.* The cost of hardware, software, and maintenance agreements is included in a submission, which enables evaluations based on price-performance as well as high performance.
- *The dataset generally must scale in size as the throughput increases*. The benchmarks are trying to model real systems, in which the demand on the system and the size of the data stored in it increase together. It makes no sense, for example, to have thousands of people per minute access hundreds of bank accounts.
- *The benchmark results are audited*. Before results can be submitted, they must be approved by a certified TPC auditor, who enforces the TPC rules that try to make sure that only fair results are submitted. Results can be challenged and disputes resolved by going before the TPC.
- *Throughput is the performance metric, but response times are limited*. For example, with TPC-C, 90% of the new order transaction response times must be less than 5 seconds.

- *An independent organization maintains the benchmarks*. Dues collected by TPC pay for an administrative structure including a chief operating office. This organization settles disputes, conducts mail ballots on approval of changes to benchmarks, holds board meetings, and so on.

SPEC System-Level File Server, Mail, and Web Benchmarks

The SPEC benchmarking effort is best known for its characterization of processor performance, but it has created benchmarks for file servers, mail servers, and Web servers.

Seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service (NFS). This benchmark was upgraded to SFS 3.0 (also called SPEC SFS97_R1) to include support for NFS version 3, using TCP in addition to UDP as the transport protocol, and making the mix of operations more realistic. Measurements on NFS systems led to a synthetic mix of reads, writes, and file operations. SFS supplies default parameters for comparative performance. For example, half of all writes are done in 8 KB blocks and half are done in partial blocks of 1, 2, or 4 KB. For reads, the mix is 85% full blocks and 15% partial blocks.

Like TPC-C, SFS scales the amount of data stored according to the reported throughput: For every 100 NFS operations per second, the capacity must increase by 1 GB. It also limits the average response time, in this case to 40 ms. Figure D.13

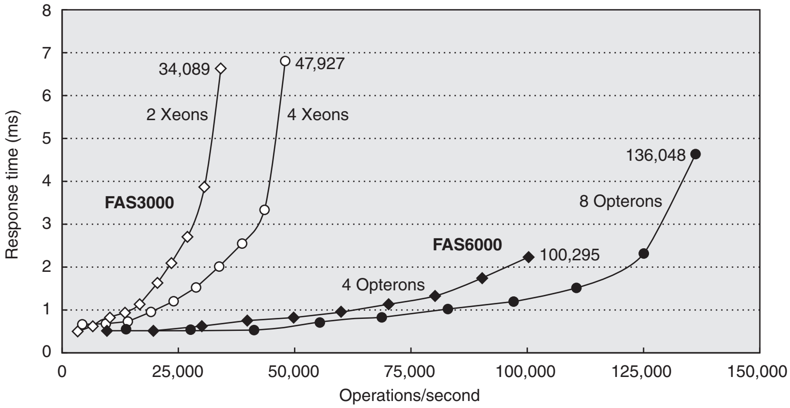


Figure D.13 SPEC SFS97_R1 performance for the NetApp FAS3050c NFS servers in two configurations. Two processors reached 34,089 operations per second and four processors did 47,927. Reported in May 2005, these systems used the Data ONTAP 7.0.1R1 operating system, 2.8 GHz Pentium Xeon microprocessors, 2 GB of DRAM per processor, 1 GB of nonvolatile memory per system, and 168 15 K RPM, 72 GB, Fibre Channel disks. These disks were connected using two or four QLogic ISP-2322 FC disk controllers.

shows average response time versus throughput for two NetApp systems. Unfortunately, unlike the TPC benchmarks, SFS does not normalize for different price configurations.

SPECMail is a benchmark to help evaluate performance of mail servers at an Internet service provider. SPECMail2001 is based on the standard Internet protocols SMTP and POP3, and it measures throughput and user response time while scaling the number of users from 10,000 to 1,000,000.

SPECWeb is a benchmark for evaluating the performance of World Wide Web servers, measuring number of simultaneous user sessions. The SPECWeb2005 workload simulates accesses to a Web service provider, where the server supports home pages for several organizations. It has three workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP).

Examples of Benchmarks of Dependability

The TPC-C benchmark does in fact have a dependability requirement. The benchmarked system must be able to handle a single disk failure, which means in practice that all submitters are running some RAID organization in their storage system.

Efforts that are more recent have focused on the effectiveness of fault tolerance in systems. Brown and Patterson [2000] proposed that availability be measured by examining the variations in system quality-of-service metrics over time as faults are injected into the system. For a Web server, the obvious metrics are performance (measured as requests satisfied per second) and degree of fault tolerance (measured as the number of faults that can be tolerated by the storage subsystem, network connection topology, and so forth).

The initial experiment injected a single fault—such as a write error in disk sector—and recorded the system's behavior as reflected in the quality-of-service metrics. The example compared software RAID implementations provided by Linux, Solaris, and Windows 2000 Server. SPECWeb99 was used to provide a workload and to measure performance. To inject faults, one of the SCSI disks in the software RAID volume was replaced with an emulated disk. It was a PC running software using a SCSI controller that appears to other devices on the SCSI bus as a disk. The disk emulator allowed the injection of faults. The faults injected included a variety of transient disk faults, such as correctable read errors, and permanent faults, such as disk media failures on writes.

Figure D.14 shows the behavior of each system under different faults. The two top graphs show Linux (on the left) and Solaris (on the right). As RAID systems can lose data if a second disk fails before reconstruction completes, the longer the reconstruction (MTTR), the lower the availability. Faster reconstruction implies decreased application performance, however, as reconstruction steals I/O resources from running applications. Thus, there is a policy choice between taking a performance hit during reconstruction or lengthening the window of vulnerability and thus lowering the predicted MTTF.

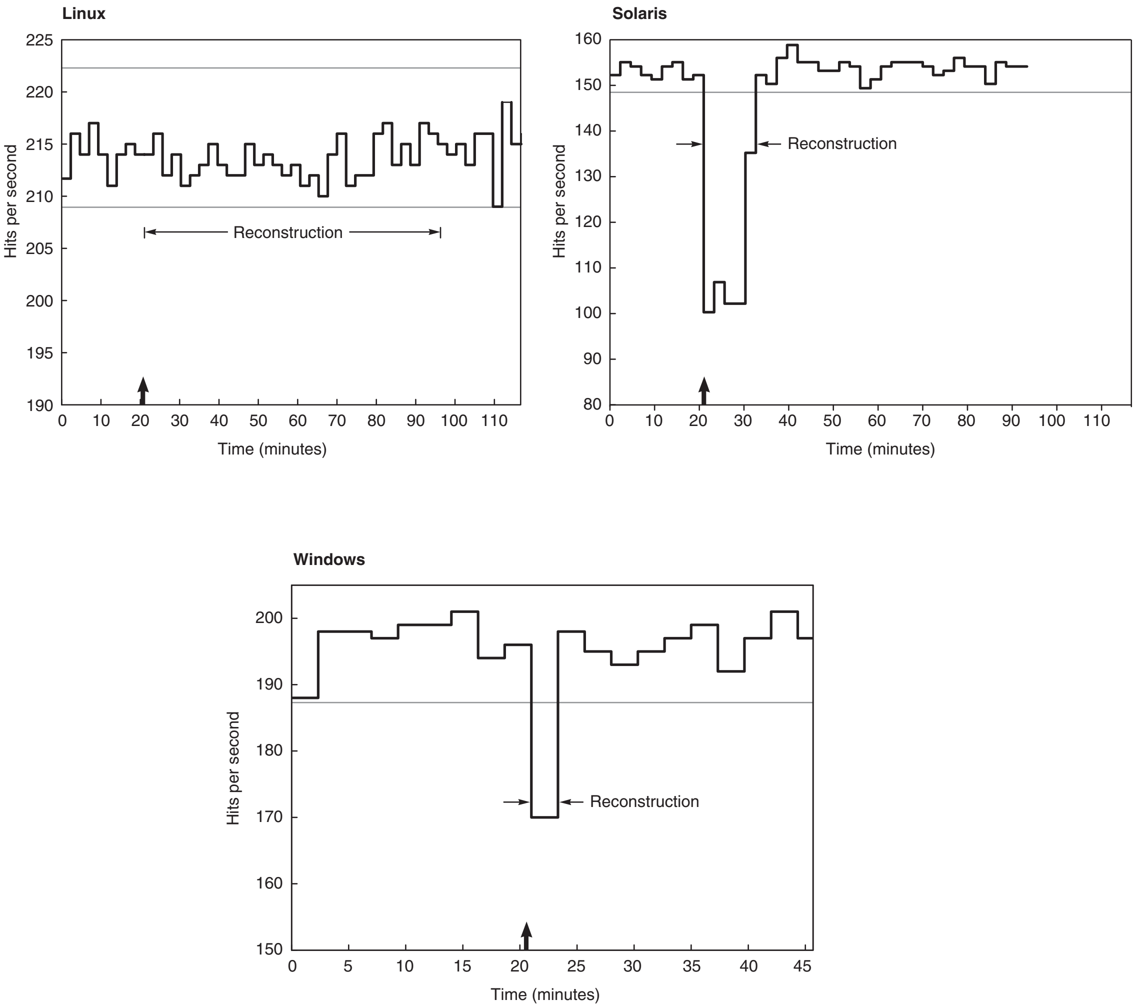


Figure D.14 Availability benchmark for software RAID systems on the same computer running Red Hat 6.0 Linux, Solaris 7, and Windows 2000 operating systems. Note the difference in philosophy on speed of reconstruction of Linux versus Windows and Solaris. The *y*-axis is behavior in hits per second running SPECWeb99. The arrow indicates time of fault insertion. The lines at the top give the 99% confidence interval of performance before the fault is inserted. A 99% confidence interval means that if the variable is outside of this range, the probability is only 1% that this value would appear.

Although none of the tested systems documented their reconstruction policies outside of the source code, even a single fault injection was able to give insight into those policies. The experiments revealed that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a failure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually. Thus, without human intervention, a Windows system that did not rebuild after a first failure remains susceptible to a second failure, which increases the window of vulnerability. It does repair quickly once told to do so.

The fault injection experiments also provided insight into other availability policies of Linux, Solaris, and Windows 2000 concerning automatic spare utilization, reconstruction rates, transient errors, and so on. Again, no system documented their policies.

In terms of managing transient faults, the fault injection experiments revealed that Linux's software RAID implementation takes an opposite approach than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus, these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient faults, ensuring that the errors are reported even if not acted upon. When faults were permanent, the systems behaved similarly.

D.5 A Little Queuing Theory

In processor design, we have simple back-of-the-envelope calculations of performance associated with the CPI formula in Chapter 1, or we can use full-scale simulation for greater accuracy at greater cost. In I/O systems, we also have a bestcase analysis as a back-of-the-envelope calculation. Full-scale simulation is also much more accurate and much more work to calculate expected performance.

With I/O systems, however, we also have a mathematical tool to guide I/O design that is a little more work and much more accurate than best-case analysis, but much less work than full-scale simulation. Because of the probabilistic nature of I/O events and because of sharing of I/O resources, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. This helpful field is called *queuing theory*. Since there are many books and courses on the subject, this section serves only as a first introduction to the topic. However, even this small amount can lead to better design of I/O systems.

Let's start with a black-box approach to I/O systems, as shown in Figure D.15. In our example, the processor is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Suppose we weren't. Although there is a mathematics that helps (Markov chains), except for a few cases, the only way to solve the resulting equations is simulation. Since the purpose of this section is to show something a little harder than back-of-the-envelope calculations but less than simulation, we won't cover such analyses here. (See the references in Appendix M for more details.)

Arrivals
Departures

Figure D.15 Treating the I/O system as a black box. This leads to a simple but important observation: If the system is in steady state, then the number of tasks entering the system must equal the number of tasks leaving the system. This *flow-balanced* state is necessary but not sufficient for steady state. If the system has been observed or measured for a sufficiently long time and mean waiting times stabilize, then we say that the system has reached steady state.

Hence, in this section we make the simplifying assumption that we are evaluating systems with multiple independent requests for I/O service that are in equilibrium: The input rate must be equal to the output rate. We also assume there is a steady supply of tasks independent for how long they wait for service. In many real systems, such as TPC-C, the task consumption rate is determined by other system characteristics, such as memory capacity.

This leads us to *Little's law,* which relates the average number of tasks in the system, the average arrival rate of new tasks, and the average time to perform a task:

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

Little's law applies to any system in equilibrium, as long as nothing inside the black box is creating new tasks or destroying them. Note that the arrival rate and the response time must use the same time unit; inconsistency in time units is a common cause of errors.

Let's try to derive Little's law. Assume we observe a system for $\text{Time}_{\text{observe}}$ minutes. During that observation, we record how long it took each task to be serviced, and then sum those times. The number of tasks completed during $\text{Time}_{\text{observe}}$ is $\text{Number}_{\text{task}}$, and the sum of the times each task spends in the system is $\text{Time}_{\text{accumulated}}$. Note that the tasks can overlap in time, so $\text{Time}_{\text{accumulated}} \geq \text{Time}_{\text{observed}}$. Then,

$$\text{Mean number of tasks in system} = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$$

$$\text{Mean response time} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$$

$$\text{Arrival rate} = \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

Algebra lets us split the first formula:

$$\frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}} \infty \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

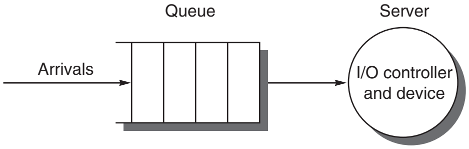


Figure D.16 The single-server model for this section. In this situation, an I/O request "departs" by being completed by the server.

If we substitute the three definitions above into this formula, and swap the resulting two terms on the right-hand side, we get Little's law:

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure D.16. The area where the tasks accumulate, waiting to be serviced, is called the *queue*, or *waiting line*. The device performing the requested service is called the *server*. Until we get to the last two pages of this section, we assume a single server.

Little's law and a series of definitions lead to several useful equations:

- $\text{Time}_{\text{server}}$—Average time to service a task; average service rate is $1/\text{Time}_{\text{server}}$, traditionally represented by the symbol μ in many queuing texts.
- $\text{Time}_{\text{queue}}$—Average time per task in the queue.
- $\text{Time}_{\text{system}}$—Average time/task in the system, or the response time, which is the sum of $\text{Time}_{\text{queue}}$ and $\text{Time}_{\text{server}}$.
- Arrival rate—Average number of arriving tasks/second, traditionally represented by the symbol λ in many queuing texts.
- $\text{Length}_{\text{server}}$—Average number of tasks in service.
- $\text{Length}_{\text{queue}}$—Average length of queue.
- $\text{Length}_{\text{system}}$—Average number of tasks in system, which is the sum of $\text{Length}_{\text{queue}}$ and $\text{Length}_{\text{server}}$.

One common misunderstanding can be made clearer by these definitions: whether the question is how long a task must wait in the queue before service starts ($\text{Time}_{\text{queue}}$) or how long a task takes until it is completed ($\text{Time}_{\text{system}}$). The latter term is what we mean by response time, and the relationship between the terms is $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$.

The mean number of tasks in service ($\text{Length}_{\text{server}}$) is simply Arrival rate × $\text{Time}_{\text{server}}$, which is Little's law. Server utilization is simply the mean number of tasks being serviced divided by the service rate. For a single server, the service rate is $1/\text{Time}_{\text{server}}$. Hence, server utilization (and, in this case, the mean number of tasks per server) is simply:

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}}$$

Service utilization must be between 0 and 1; otherwise, there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Note that this formula is just a restatement of Little's law. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many queuing theory texts.

Example Suppose an I/O system with a single disk gets on average 50 I/O requests per second. Assume the average time for a disk to service an I/O request is 10 ms. What is the utilization of the I/O system?

Answer Using the equation above, with 10 ms represented as 0.01 seconds, we get: 50

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{50}{\text{sec}} \times 0.01\,\text{sec} = 0.50$$

Therefore, the I/O system utilization is 0.5.

How the queue delivers tasks to the server is called the *queue discipline*. The simplest and most common discipline is *first in, first out* (FIFO). If we assume FIFO, we can relate time waiting in the queue to the mean number of tasks in the queue:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \begin{array}{l}\text{Mean time to complete service of task when}\\ \text{new task arrives if server is busy}\end{array}$$

That is, the time in the queue is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever task is being serviced when a new task arrives. (There is one more restriction about the arrival of tasks, which we reveal on page D-28.)

The last component of the equation is not as simple as it first appears. A new task can arrive at any instant, so we have no basis to know how long the existing task has been in the server. Although such requests are random events, if we know something about the distribution of events, we can predict performance.

Poisson Distribution of Random Variables

To estimate the last component of the formula we need to know a little about distributions of *random variables*. A variable is random if it takes one of a specified set of values with a specified probability; that is, you cannot know exactly what its next value will be, but you may know the probability of all possible values.

Requests for service from an I/O system can be modeled by a random variable because the operating system is normally switching between several processes that generate independent I/O requests. We also model I/O service times by a random variable given the probabilistic nature of disks in terms of seek and rotational delays.

One way to characterize the distribution of values of a random variable with discrete values is a *histogram*, which divides the range between the minimum and maximum values into subranges called *buckets*. Histograms then plot the number in each bucket as columns.

Histograms work well for distributions that are discrete values—for example, the number of I/O requests. For distributions that are not discrete values, such as time waiting for an I/O request, we have two choices. Either we need a curve to plot the values over the full range, so that we can estimate accurately the value, or we need a very fine time unit so that we get a very large number of buckets to estimate time accurately. For example, a histogram can be built of disk service times measured in intervals of 10 μs although disk service times are truly continuous.

Hence, to be able to solve the last part of the previous equation we need to characterize the distribution of this random variable. The mean time and some measure of the variance are sufficient for that characterization.

For the first term, we use the *weighted arithmetic mean time*. Let's first assume that after measuring the number of occurrences, say, n_i, of tasks, you could compute frequency of occurrence of task i:

$$f_i = \frac{n_i}{\left(\sum_{i=1}^{n} n_i\right)}$$

Then weighted arithmetic mean is

$$\text{Weighted arithmetic mean time} = f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n$$

where T_i is the time for task i and f_i is the frequency of occurrence of task i.

To characterize variability about the mean, many people use the standard deviation. Let's use the *variance* instead, which is simply the square of the standard deviation, as it will help us with characterizing the probability distribution. Given the weighted arithmetic mean, the variance can be calculated as

$$\text{Variance} = \left(f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2\right) - \text{Weighted arithmetic mean time}^2$$

It is important to remember the units when computing variance. Let's assume the distribution is of time. If time is about 100 milliseconds, then squaring it yields 10,000 square milliseconds. This unit is certainly unusual. It would be more convenient if we had a unitless measure.

To avoid this unit problem, we use the *squared coefficient of variance*, traditionally called C^2:

$$C^2 = \frac{\text{Variance}}{\text{Weighted arithmetic mean time}^2}$$

We can solve for C, the coefficient of variance, as

$$C = \frac{\sqrt{\text{Variance}}}{\text{Weighted arithmetic mean time}} = \frac{\text{Standard deviation}}{\text{Weighted arithmetic mean time}}$$

We are trying to characterize random events, but to be able to predict performance we need a distribution of random events where the mathematics is tractable. The most popular such distribution is the *exponential distribution*, which has a C value of 1.

Note that we are using a constant to characterize variability about the mean. The invariance of C over time reflects the property that the history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless*, and this property is an important assumption used to predict behavior using these models. (Suppose this memoryless property did not exist; then, we would have to worry about the exact arrival times of requests relative to each other, which would make the mathematics considerably less tractable!)

One of the most widely used exponential distributions is called a *Poisson distribution*, named after the mathematician Siméon Poisson. It is used to characterize random events in a given time interval and has several desirable mathematical properties. The Poisson distribution is described by the following equation (called the probability mass function):

$$\text{Probability}(k) = \frac{e^{-a} \times a^k}{k!}$$

where $a = \text{Rate of events} \times \text{Elapsed time}$. If interarrival times are exponentially distributed and we use the arrival rate from above for rate of events, the number of arrivals in a time interval t is a *Poisson process*, which has the Poisson distribution with $a = \text{Arrival rate} \times t$. As mentioned on page D-26, the equation for $\text{Time}_{\text{server}}$ has another restriction on task arrival: It holds only for Poisson processes.

Finally, we can answer the question about the length of time a new task must wait for the server to complete a task, called the *average residual service time*, which again assumes Poisson arrivals:

$$\text{Average residual service time} = 1/2 \times \text{Arithemtic mean} \times \left(1 + \text{C}^2\right)$$

Although we won't derive this formula, we can appeal to intuition. When the distribution is not random and all possible values are equal to the average, the standard deviation is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect. If the distribution is random and it is Poisson, then C is 1 and the average residual service time equals the weighted arithmetic mean time.

Example Using the definitions and formulas above, derive the average time waiting in the queue ($\text{Time}_{\text{queue}}$) in terms of the average service time ($\text{Time}_{\text{server}}$) and server utilization.

Answer All tasks in the queue ($\text{Length}_{\text{queue}}$) ahead of the new task must be completed before the task can be serviced; each takes on average $\text{Time}_{\text{server}}$. If a task is at the server, it takes average residual service time to complete. The chance the server is busy is *server utilization*; hence, the expected time for service is Server utilization × Average residual service time. This leads to our initial formula:

$$\begin{aligned}\text{Time}_{\text{queue}} = {} & \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} \\ & + \text{Server utilization} \times \text{Average residual service time}\end{aligned}$$

Replacing the average residual service time by its definition and $\text{Length}_{\text{queue}}$ by $\text{Arrival rate} \times \text{Time}_{\text{queue}}$ yields

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times \left[1/2 \times \text{Time}_{\text{server}} \times \left(1 + C^2\right)\right] + \left(\text{Arrival rate} \times \text{Time}_{\text{queue}}\right) \times \text{Time}_{\text{server}}$$

Since this section is concerned with exponential distributions, C^2 is 1. Thus

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times \text{Time}_{\text{server}} + \left(\text{Arrival rate} \times \text{Time}_{\text{queue}}\right) \times \text{Time}_{\text{server}}$$

Rearranging the last term, let us replace Arrival rate $\times$ $\text{Time}_{\text{server}}$ by Server utilization:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}} + (\text{Arrival rate} \times \text{Time}_{\text{server}}) \times \text{Time}_{\text{queue}} \\ &= \text{Server utilization} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Time}_{\text{queue}}\end{aligned}$$

Rearranging terms and simplifying gives us the desired equation:

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Time}_{\text{queue}}$$

$$\text{Time}_{\text{queue}} - \text{Server utilization} \times \text{Time}_{\text{queue}} = \text{Server utilization} \times \text{Time}_{\text{server}}$$

$$\text{Time}_{\text{queue}} \times (1 - \text{Server utilization}) = \text{Server utilization} \times \text{Time}_{\text{server}}$$

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}$$

Little's law can be applied to the components of the black box as well, since they must also be in equilibrium:

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{queue}}$$

If we substitute for $\text{Time}_{\text{queue}}$ from above, we get:

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}$$

Since Arrival rate $\times$ $\text{Time}_{\text{server}}$ = Server utilization, we can simplify further:

$$\text{Length}_{\text{queue}} = \text{Server utilization} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = \frac{\text{Server utilization}^2}{(1 - \text{Server utilization})}$$

This relates number of items in queue to service utilization.

Example For the system in the example on page D-26, which has a server utilization of 0.5, what is the mean number of I/O requests in the queue?

Answer Using the equation above,

$$\text{Length}_{\text{queue}} = \frac{\text{Server utilization}^2}{(1 - \text{Server utilization})} = \frac{0.5^2}{(1 - 0.5)} = \frac{0.25}{0.50} = 0.5$$

Therefore, there are 0.5 requests on average in the queue.

As mentioned earlier, these equations and this section are based on an area of applied mathematics called *queuing theory*, which offers equations to predict behavior of such random variables. Real systems are too complex for queuing theory to provide exact analysis, hence queuing theory works best when only approximate answers are needed.

Queuing theory makes a sharp distinction between past events, which can be characterized by measurements using simple arithmetic, and future events, which are predictions requiring more sophisticated mathematics. In computer systems, we commonly predict the future from the past; one example is least recently used block replacement (see Chapter 2). Hence, the distinction between measurements and predicted distributions is often blurred; we use measurements to verify the type of distribution and then rely on the distribution thereafter.

Let's review the assumptions about the queuing model:

- The system is in equilibrium.
- The times between two successive requests arriving, called the *interarrival times*, are exponentially distributed, which characterizes the arrival rate mentioned earlier.
- The number of sources of requests is unlimited. (This is called an *infinite population model* in queuing theory; finite population models are used when arrival rates vary with the number of jobs already in the system.)
- The server can start on the next job immediately after finishing the prior one.
- There is no limit to the length of the queue, and it follows the first in, first out order discipline, so all tasks in line must be completed.
- There is one server.

Such a queue is called *M/M/1:*

M = exponentially random request arrival ($C^2 = 1$), with *M* standing for A. A. Markov, the mathematician who defined and analyzed the memoryless processes mentioned earlier

M = exponentially random service time ($C^2 = 1$), with *M* again for Markov

1 = single server

The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for three reasons—one good, one fair, and one bad. The good reason is that a superposition of many arbitrary distributions acts as an exponential distribution. Many times in computer systems, a particular behavior is the result of many components interacting, so an exponential distribution of interarrival times is the right model. The fair reason is that when variability is unclear, an exponential distribution with intermediate variability ($C = 1$) is a safer guess than low variability ($C \approx 0$) or high variability (large C). The bad reason is that the math is simpler if you assume exponential distributions.

Let's put queuing theory to work in a few examples.

Example Suppose a processor sends 40 disk I/Os per second, these requests are exponentially distributed, and the average service time of an older disk is 20 ms. Answer the following questions:

1. On average, how utilized is the disk?
2. What is the average time spent in the queue?
3. What is the average response time for a disk request, including the queuing time and disk service time?

Answer Let's restate these facts:

Average number of arriving tasks/second is 40.

Average disk time to service a task is 20 ms (0.02 sec).

The server utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.02 = 0.8$$

Since the service times are exponentially distributed, we can use the simplified formula for the average time spent waiting in line:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} \\ &= 20 \text{ ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80 \text{ ms}\end{aligned}$$

The average response time is

$$\text{Time system} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 \text{ ms} = 100 \text{ ms}$$

Thus, on average we spend 80% of our time waiting in the queue!

Example Suppose we get a new, faster disk. Recalculate the answers to the questions above, assuming the disk service time is 10 ms.

Answer The disk utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.01 = 0.4$$

The formula for the average time spent waiting in line:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} \\ &= 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms}\end{aligned}$$

The average response time is $10 + 6.7$ ms or 16.7 ms, 6.0 times faster than the old response time even though the new service time is only 2.0 times faster.

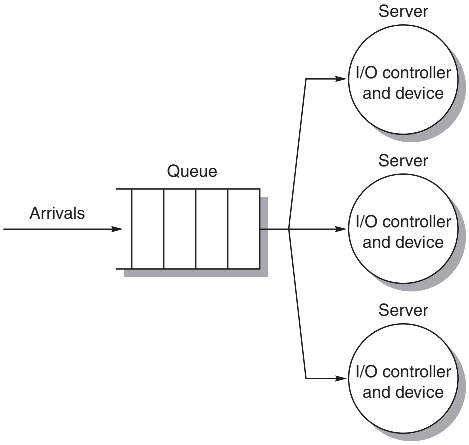

Server
I/O controller and device
Server
I/O controller and device
Server
I/O controller and device
Queue
Arrivals

Figure D.17 The M/M/m multiple-server model.

Thus far, we have been assuming a single server, such as a single disk. Many real systems have multiple disks and hence could use multiple servers, as in Figure D.17. Such a system is called an *M/M/m* model in queuing theory.

Let's give the same formulas for the M/M/m queue, using $\mathrm{N_{servers}}$ to represent the number of servers. The first two formulas are easy:

$$\begin{aligned}\text{Utilization} &= \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{\text{N}_{\text{servers}}}\\ \text{Length}_{\text{queue}} &= \text{Arrival rate} \times \text{Time}_{\text{queue}}\end{aligned}$$

The time waiting in the queue is

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{P}_{\text{tasks} \geq \text{N}_{\text{servers}}}}{\text{N}_{\text{servers}} \times (1 - \text{Utilization})}$$

This formula is related to the one for M/M/1, except we replace utilization of a single server with the probability that a task will be queued as opposed to being immediately serviced, and divide the time in queue by the number of servers. Alas, calculating the probability of jobs being in the queue is much more complicated when there are $\mathrm{N_{servers}}$. First, the probability that there are no tasks in the system is

$$\text{Prob}_{0\text{ tasks}} = \left[1 + \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} + \sum_{n=1}^{\text{N}_{\text{servers}}-1} \frac{(\text{N}_{\text{servers}} \times \text{Utilization})^{n}}{n!}\right]^{-1}$$

Then the probability there are as many or more tasks than we have servers is

$$\text{Prob}_{\text{tasks} \geq \text{N}_{\text{servers}}} = \frac{\text{N}_{\text{servers}} \times \text{Utilization}^{\text{N}_{\text{servers}}}}{\text{N}_{\text{servers}}! \times (1 - \text{Utilization})} \times \text{Prob}_{0\text{ tasks}}$$

Note that if $N_{servers}$ is 1, $Prob_{task \geq N_{servers}}$ simplifies back to Utilization, and we get the same formula as for M/M/1. Let's try an example.

Example Suppose instead of a new, faster disk, we add a second slow disk and duplicate the data so that reads can be serviced by either disk. Let's assume that the requests are all reads. Recalculate the answers to the earlier questions, this time using an M/M/m queue.

Answer The average utilization of the two disks is then

$$\text{Server utilization} = \frac{\text{Arrival rate} \times \text{Time}_{server}}{N_{servers}} = \frac{40 \times 0.02}{2} = 0.4$$

We first calculate the probability of no tasks in the queue:

$$\begin{aligned}\text{Prob}_{0\ tasks} &= \left[1 + \frac{(2 \times \text{Utilization})^2}{2! \times (1 - \text{Utilization})} + \sum_{n=1}^{1} \frac{(2 \times \text{Utilization})^n}{n!}\right]^{-1} \\ &= \left[1 + \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} + (2 \times 0.4)\right]^{-1} = \left[1 + \frac{0.640}{1.2} + 0.800\right]^{-1} \\ &= [1 + 0.533 + 0.800]^{-1} = 2.333^{-1}\end{aligned}$$

We use this result to calculate the probability of tasks in the queue:

$$\begin{aligned}\text{Prob}_{tasks \geq N_{servers}} &= \frac{2 \times \text{Utilization}^2}{2! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ tasks} \\ &= \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} \times 2.333^{-1} = \frac{0.640}{1.2} \times 2.333^{-1} \\ &= 0.533/2.333 = 0.229\end{aligned}$$

Finally, the time waiting in the queue:

$$\begin{aligned}\text{Time}_{queue} &= \text{Time}_{server} \times \frac{\text{Prob}_{tasks \geq N_{servers}}}{N_{servers} \times (1 - \text{Utilization})} \\ &= 0.020 \times \frac{0.229}{2 \times (1 - 0.4)} = 0.020 \times \frac{0.229}{1.2} \\ &= 0.020 \times 0.190 = 0.0038\end{aligned}$$

The average response time is 20 + 3.8 ms or 23.8 ms. For this workload, two disks cut the queue waiting time by a factor of 21 over a single slow disk and a factor of 1.75 versus a single fast disk. The mean service time of a system with a single fast disk, however, is still 1.4 times faster than one with two disks since the disk service time is 2.0 times faster.

It would be wonderful if we could generalize the M/M/m model to multiple queues and multiple servers, as this step is much more realistic. Alas, these models are very hard to solve and to use, and so we won't cover them here.

D.6 Crosscutting Issues

Point-to-Point Links and Switches Replacing Buses

Point-to-point links and switches are increasing in popularity as Moore's law continues to reduce the cost of components. Combined with the higher I/O bandwidth demands from faster processors, faster disks, and faster local area networks, the decreasing cost advantage of buses means the days of buses in desktop and server computers are numbered. This trend started in high-performance computers in the last edition of the book, and by 2011 has spread itself throughout storage. Figure D.18 shows the old bus-based standards and their replacements.

The number of bits and bandwidth for the new generation is per direction, so they double for both directions. Since these new designs use many fewer wires, a common way to increase bandwidth is to offer versions with several times the number of wires and bandwidth.

Block Servers versus Filers

Thus far, we have largely ignored the role of the operating system in storage. In a manner analogous to the way compilers use an instruction set, operating systems determine what I/O techniques implemented by the hardware will actually be used. The operating system typically provides the file abstraction on top of blocks stored on the disk. The terms *logical units*, *logical volumes*, and *physical volumes* are related terms used in Microsoft and UNIX systems to refer to subset collections of disk blocks.

Standard	Width (bits)	Length (meters)	Clock rate	MB/sec	Max I/O devices
(Parallel) ATA	8	0.5	133 MHz	133	2
Serial ATA	2	2	3 GHz	300	?
SCSI	16	12	80 MHz	320	15
Serial Attach SCSI	1	10	(DDR)	375	16,256
PCI	32/64	0.5	33/66 MHz	533	?
PCI Express	2	0.5	3 GHz	250	?

Figure D.18 Parallel I/O buses and their point-to-point replacements. Note the bandwidth and wires are per direction, so bandwidth doubles when sending both directions.

A logical unit is the element of storage exported from a disk array, usually constructed from a subset of the array's disks. A logical unit appears to the server as a single virtual "disk." In a RAID disk array, the logical unit is configured as a particular RAID layout, such as RAID 5. A physical volume is the device file used by the file system to access a logical unit. A logical volume provides a level of virtualization that enables the file system to split the physical volume across multiple pieces or to stripe data across multiple physical volumes. A logical unit is an abstraction of a disk array that presents a virtual disk to the operating system, while physical and logical volumes are abstractions used by the operating system to divide these virtual disks into smaller, independent file systems.

Having covered some of the terms for collections of blocks, we must now ask: Where should the file illusion be maintained: in the server or at the other end of the storage area network?

The traditional answer is the server. It accesses storage as disk blocks and maintains the metadata. Most file systems use a file cache, so the server must maintain consistency of file accesses. The disks may be *direct attached*—found inside a server connected to an I/O bus—or attached over a storage area network, but the server transmits data blocks to the storage subsystem.

The alternative answer is that the disk subsystem itself maintains the file abstraction, and the server uses a file system protocol to communicate with storage. Example protocols are Network File System (NFS) for UNIX systems and Common Internet File System (CIFS) for Windows systems. Such devices are called *network attached storage* (NAS) devices since it makes no sense for storage to be directly attached to the server. The name is something of a misnomer because a storage area network like FC-AL can also be used to connect to block servers. The term *filer* is often used for NAS devices that only provide file service and file storage. Network Appliance was one of the first companies to make filers.

The driving force behind placing storage on the network is to make it easier for many computers to share information and for operators to maintain the shared system.

Asynchronous I/O and Operating Systems

Disks typically spend much more time in mechanical delays than in transferring data. Thus, a natural path to higher I/O performance is parallelism, trying to get many disks to simultaneously access data for a program.

The straightforward approach to I/O is to request data and then start using it. The operating system then switches to another process until the desired data arrive, and then the operating system switches back to the requesting process. Such a style is called *synchronous I/O*—the process waits until the data have been read from disk.

The alternative model is for the process to continue after making a request, and it is not blocked until it tries to read the requested data. Such *asynchronous I/O* allows the process to continue making requests so that many I/O requests can be operating simultaneously. Asynchronous I/O shares the same philosophy as caches in out-of-order CPUs, which achieve greater bandwidth by having multiple outstanding events.

D.7 Designing and Evaluating an I/O System— The Internet Archive Cluster

The art of I/O system design is to find a design that meets goals for cost, dependability, and variety of devices while avoiding bottlenecks in I/O performance and dependability. Avoiding bottlenecks means that components must be balanced between main memory and the I/O device, because performance and dependability—and hence effective cost-performance or cost-dependability—can only be as good as the weakest link in the I/O chain. The architect must also plan for expansion so that customers can tailor the I/O to their applications. This expansibility, both in numbers and types of I/O devices, has its costs in longer I/O buses and networks, larger power supplies to support I/O devices, and larger cabinets.

In designing an I/O system, we analyze performance, cost, capacity, and availability using varying I/O connection schemes and different numbers of I/O devices of each type. Here is one series of steps to follow in designing an I/O system. The answers for each step may be dictated by market requirements or simply by cost, performance, and availability goals.

1. List the different types of I/O devices to be connected to the machine, or list the standard buses and networks that the machine will support.
2. List the physical requirements for each I/O device. Requirements include size, power, connectors, bus slots, expansion cabinets, and so on.
3. List the cost of each I/O device, including the portion of cost of any controller needed for this device.
4. List the reliability of each I/O device.
5. Record the processor resource demands of each I/O device. This list should include:
   - Clock cycles for instructions used to initiate an I/O, to support operation of an I/O device (such as handling interrupts), and to complete I/O
   - Processor clock stalls due to waiting for I/O to finish using the memory, bus, or cache
   - Processor clock cycles to recover from an I/O activity, such as a cache flush
6. List the memory and I/O bus resource demands of each I/O device. Even when the processor is not using memory, the bandwidth of main memory and the I/O connection is limited.

7. The final step is assessing the performance and availability of the different ways to organize these I/O devices. When you can afford it, try to avoid single points of failure. Performance can only be properly evaluated with simulation, although it may be estimated using queuing theory. Reliability can be calculated assuming I/O devices fail independently and that the times to failure are exponentially distributed. Availability can be computed from reliability by estimating MTTF for the devices, taking into account the time from failure to repair.

Given your cost, performance, and availability goals, you then select the best organization.

Cost-performance goals affect the selection of the I/O scheme and physical design. Performance can be measured either as megabytes per second or I/Os per second, depending on the needs of the application. For high performance, the only limits should be speed of I/O devices, number of I/O devices, and speed of memory and processor. For low cost, most of the cost should be the I/O devices themselves. Availability goals depend in part on the cost of unavailability to an organization.

Rather than create a paper design, let's evaluate a real system.

The Internet Archive Cluster

To make these ideas clearer, we'll estimate the cost, performance, and availability of a large storage-oriented cluster at the Internet Archive. The Internet Archive began in 1996 with the goal of making a historical record of the Internet as it changed over time. You can use the Wayback Machine interface to the Internet Archive to perform time travel to see what the Web site at a URL looked like sometime in the past. It contains over a petabyte (10^{15} bytes) and is growing by 20 terabytes (10^{12} bytes) of new data per month, so expansible storage is a requirement. In addition to storing the historical record, the same hardware is used to crawl the Web every few months to get snapshots of the Internet.

Clusters of computers connected by local area networks have become a very economical computation engine that works well for some applications. Clusters also play an important role in Internet services such the Google search engine, where the focus is more on storage than it is on computation, as is the case here.

Although it has used a variety of hardware over the years, the Internet Archive is moving to a new cluster to become more efficient in power and in floor space. The basic building block is a 1U storage node called the PetaBox GB2000 from Capricorn Technologies. In 2006, it used four 500 GB Parallel ATA (PATA) disk drives, 512 MB of DDR266 DRAM, one 10/100/1000 Ethernet interface, and a 1 GHz C3 processor from VIA, which executes the 80x86 instruction set. This node dissipates about 80 watts in typical configurations.

Figure D.19 shows the cluster in a standard VME rack. Forty of the GB2000s fit in a standard VME rack, which gives the rack 80 TB of raw capacity. The 40 nodes are connected together with a 48-port 10/100 or 10/100/1000 switch, and it

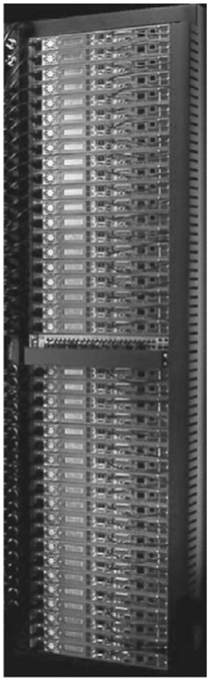


Figure D.19 The TB-80 VME rack from Capricorn Systems used by the Internet Archive. All cables, switches, and displays are accessible from the front side, and the back side is used only for airflow. This allows two racks to be placed back-to-back, which reduces the floor space demands in machine rooms.

dissipates about 3 KW. The limit is usually 10 KW per rack in computer facilities, so it is well within the guidelines.

A petabyte needs 12 of these racks, connected by a higher-level switch that connects the Gbit links coming from the switches in each of the racks.

Estimating Performance, Dependability, and Cost of the Internet Archive Cluster

To illustrate how to evaluate an I/O system, we'll make some guesses about the cost, performance, and reliability of the components of this cluster. We make the following assumptions about cost and performance:

- The VIA processor, 512 MB of DDR266 DRAM, ATA disk controller, power supply, fans, and enclosure cost $500.
- Each of the four 7200 RPM Parallel ATA drives holds 500 GB, has an average time seek of 8.5 ms, transfers at 50 MB/sec from the disk, and costs $375. The PATA link speed is 133 MB/sec.

- The 48-port 10/100/1000 Ethernet switch and all cables for a rack cost $3000.
- The performance of the VIA processor is 1000 MIPS.
- The ATA controller adds 0.1 ms of overhead to perform a disk I/O.
- The operating system uses 50,000 CPU instructions for a disk I/O.
- The network protocol stacks use 100,000 CPU instructions to transmit a data block between the cluster and the external world.
- The average I/O size is 16 KB for accesses to the historical record via the Wayback interface, and 50 KB when collecting a new snapshot.

Example Evaluate the cost per I/O per second (IOPS) of the 80 TB rack. Assume that every disk I/O requires an average seek and average rotational delay. Assume that the workload is evenly divided among all disks and that all devices can be used at 100% of capacity; that is, the system is limited only by the weakest link, and it can operate that link at 100% utilization. Calculate for both average I/O sizes.

Answer I/O performance is limited by the weakest link in the chain, so we evaluate the maximum performance of each link in the I/O chain for each organization to determine the maximum performance of that organization.

Let's start by calculating the maximum number of IOPS for the CPU, main memory, and I/O bus of one GB2000. The CPU I/O performance is determined by the speed of the CPU and the number of instructions to perform a disk I/O and to send it over the network:

$$\begin{aligned}\text{Maximum IOPS for CPU} &= \frac{1000\text{ MIPS}}{50{,}000\text{ instructions per I/O} + 100{,}000\text{ instructions per message}}\\ &= 6667\text{ IOPS}\end{aligned}$$

The maximum performance of the memory system is determined by the memory bandwidth and the size of the I/O transfers:

$$\text{Maximum IOPS for main memory} = \frac{266 \times 8}{16\text{ KB per I/O}} \approx 133{,}000\text{ IOPS}$$

$$\text{Maximum IOPS for main memory} = \frac{266 \times 8}{50\text{ KB per I/O}} \approx 42{,}500\text{ IOPS}$$

The Parallel ATA link performance is limited by the bandwidth and the size of the I/O:

$$\text{Maximum IOPS for the I/O bus} = \frac{133\text{ MB/sec}}{16\text{ KB per I/O}} \approx 8300\text{ IOPS}$$

$$\text{Maximum IOPS for the I/O bus} = \frac{133\text{ MB/sec}}{50\text{ KB per I/O}} \approx 2700\text{ IOPS}$$

Since the box has two buses, the I/O bus limits the maximum performance to no more than 18,600 IOPS for 16 KB blocks and 5400 IOPS for 50 KB blocks.

Now it's time to look at the performance of the next link in the I/O chain, the ATA controllers. The time to transfer a block over the PATA channel is

$$\text{Parallel ATA transfer time} = \frac{16\ \text{KB}}{133\ \text{MB/sec}} \approx 0.1\ \text{ms}$$

$$\text{Parallel ATA transfer time} = \frac{50\ \text{KB}}{133\ \text{MB/sec}} \approx 0.4\ \text{ms}$$

Adding the 0.1 ms ATA controller overhead means 0.2 ms to 0.5 ms per I/O, making the maximum rate per controller

$$\text{Maximum IOPS per ATA controller} = \frac{1}{0.2\ \text{ms}} = 5000\ \text{IOPS}$$

$$\text{Maximum IOPS per ATA controller} = \frac{1}{0.5\ \text{ms}} = 2000\ \text{IOPS}$$

The next link in the chain is the disks themselves. The time for an average disk I/O is

$$\text{I/O time} = 8.5\ \text{ms} + \frac{0.5}{7200\ \text{RPM}} + \frac{16\ \text{KB}}{50\ \text{MB/sec}} = 8.5 + 4.2 + 0.3 = 13.0\ \text{ms}$$

$$\text{I/O time} = 8.5\ \text{ms} + \frac{0.5}{7200\ \text{RPM}} + \frac{50\ \text{KB}}{50\ \text{MB/sec}} = 8.5 + 4.2 + 1.0 = 13.7\ \text{ms}$$

Therefore, disk performance is

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{13.0\ \text{ms}} \approx 77\ \text{IOPS}$$

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{13.7\ \text{ms}} \approx 73\ \text{IOPS}$$

or 292 to 308 IOPS for the four disks.

The final link in the chain is the network that connects the computers to the outside world. The link speed determines the limit:

$$\text{Maximum IOPS per 1000 Mbit Ethernet link} = \frac{1000\ \text{Mbit}}{16\ \text{K} \times 8} = 7812\ \text{IOPS}$$

$$\text{Maximum IOPS per 1000 Mbit Ethernet link} = \frac{1000\ \text{Mbit}}{50\ \text{K} \times 8} = 2500\ \text{IOPS}$$

Clearly, the performance bottleneck of the GB2000 is the disks. The IOPS for the whole rack is 40 × 308 or 12,320 IOPS to 40 × 292 or 11,680 IOPS. The network switch would be the bottleneck if it couldn't support 12,320 × 16 K × 8 or 1.6 Gbits/sec for 16 KB blocks and 11,680 × 50 K × 8 or 4.7 Gbits/sec for 50 KB blocks. We assume that the extra 8 Gbit ports of the 48-port switch connects the rack to the rest of the world, so it could support the full IOPS of the collective 160 disks in the rack.

Using these assumptions, the cost is 40 × ($500 + 4 × $375) + $3000 + $1500 or $84,500 for an 80 TB rack. The disks themselves are almost 60% of the cost. The cost per terabyte is almost $1000, which is about a factor of 10 to 15 better than storage cluster from the prior edition in 2001. The cost per IOPS is about $7.

Calculating MTTF of the TB-80 Cluster

Internet services such as Google rely on many copies of the data at the application level to provide dependability, often at different geographic sites to protect against environmental faults as well as hardware faults. Hence, the Internet Archive has two copies of the data in each site and has sites in San Francisco, Amsterdam, and Alexandria, Egypt. Each site maintains a duplicate copy of the high-value content—music, books, film, and video—and a single copy of the historical Web crawls. To keep costs low, there is no redundancy in the 80 TB rack.

Example Let's look at the resulting mean time to fail of the rack. Rather than use the manufacturer's quoted MTTF of 600,000 hours, we'll use data from a recent survey of disk drives [Gray and van Ingen 2005]. As mentioned in Chapter 1, about 3% to 7% of ATA drives fail per year, for an MTTF of about 125,000 to 300,000 hours. Make the following assumptions, again assuming exponential lifetimes:

- CPU/memory/enclosure MTTF is 1,000,000 hours.
- PATA Disk MTTF is 125,000 hours.
- PATA controller MTTF is 500,000 hours.
- Ethernet Switch MTTF is 500,000 hours.
- Power supply MTTF is 200,000 hours.
- Fan MTTF is 200,000 hours.
- PATA cable MTTF is 1,000,000 hours.

Answer Collecting these together, we compute these failure rates:

$$\text{Failure rate} = \frac{40}{1{,}000{,}000} + \frac{160}{125{,}000} + \frac{40}{500{,}000} + \frac{1}{500{,}000} + \frac{40}{200{,}000} + \frac{40}{200{,}000} + \frac{80}{1{,}000{,}000}$$

$$= \frac{40 + 1280 + 80 + 2 + 200 + 200 + 80}{1{,}000{,}000 \text{ hours}} = \frac{1882}{1{,}000{,}000 \text{ hours}}$$

The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF} = \frac{1}{\text{Failure rate}} = \frac{1{,}000{,}000 \text{ hours}}{1882} = 531 \text{ hours}$$

That is, given these assumptions about the MTTF of components, something in a rack fails on average every 3 weeks. About 70% of the failures would be the disks, and about 20% would be fans or power supplies.

D.8 Putting It All Together: NetApp FAS6000 Filer

Network Appliance entered the storage market in 1992 with a goal of providing an easy-to-operate file server running NSF using their own log-structured file system and a RAID 4 disk array. The company later added support for the Windows CIFS file system and a RAID 6 scheme called *row-diagonal parity* or *RAID-DP* (see page D-8). To support applications that want access to raw data blocks without the overhead of a file system, such as database systems, NetApp filers can serve data blocks over a standard Fibre Channel interface. NetApp also supports *iSCSI*, which allows SCSI commands to run over a TCP/IP network, thereby allowing the use of standard networking gear to connect servers to storage, such as Ethernet, and hence at a greater distance.

The latest hardware product is the FAS6000. It is a multiprocessor based on the AMD Opteron microprocessor connected using its HyperTransport links. The microprocessors run the NetApp software stack, including NSF, CIFS, RAID-DP, SCSI, and so on. The FAS6000 comes as either a dual processor (FAS6030) or a quad processor (FAS6070). As mentioned in Chapter 5, DRAM is distributed to each microprocessor in the Opteron. The FAS6000 connects 8 GB of DDR2700 to each Opteron, yielding 16 GB for the FAS6030 and 32 GB for the FAS6070. As mentioned in Chapter 4, the DRAM bus is 128 bits wide, plus extra bits for SEC/DED memory. Both models dedicate four HyperTransport links to I/O.

As a filer, the FAS6000 needs a lot of I/O to connect to the disks and to connect to the servers. The integrated I/O consists of:

- 8 Fibre Channel (FC) controllers and ports
- 6 Gigabit Ethernet links
- 6 slots for x8 (2 GB/sec) PCI Express cards
- 3 slots for PCI-X 133 MHz, 64-bit cards
- Standard I/O options such as IDE, USB, and 32-bit PCI

The 8 Fibre Channel controllers can each be attached to 6 shelves containing 14 3.5-inch FC disks. Thus, the maximum number of drives for the integrated I/O is $8 \times 6 \times 14$ or 672 disks. Additional FC controllers can be added to the option slots to connect up to 1008 drives, to reduce the number of drives per FC network so as to reduce contention, and so on. At 500 GB per FC drive, if we assume the RAID RDP group is 14 data disks and 2 check disks, the available data capacity is 294 TB for 672 disks and 441 TB for 1008 disks.

It can also connect to Serial ATA disks via a Fibre Channel to SATA bridge controller, which, as its name suggests, allows FC and SATA to communicate.

The six 1-gigabit Ethernet links connect to servers to make the FAS6000 look like a file server if running NTFS or CIFS or like a block server if running iSCSI.

For greater dependability, FAS6000 filers can be paired so that if one fails, the other can take over. Clustered failover requires that both filers have access to all disks in the pair of filers using the FC interconnect. This interconnect also allows each filer to have a copy of the log data in the NVRAM of the other filer and to keep the clocks of the pair synchronized. The health of the filers is constantly monitored, and failover happens automatically. The healthy filer maintains its own network identity and its own primary functions, but it also assumes the network identity of the failed filer and handles all its data requests via a virtual filer until an administrator restores the data service to the original state.

D.9 Fallacies and Pitfalls

Fallacy *Components fail fast*

A good deal of the fault-tolerant literature is based on the simplifying assumption that a component operates perfectly until a latent error becomes effective, and then a failure occurs that stops the component.

The Tertiary Disk project had the opposite experience. Many components started acting strangely long before they failed, and it was generally up to the system operator to determine whether to declare a component as failed. The component would generally be willing to continue to act in violation of the service agreement until an operator "terminated" that component.

Figure D.20 shows the history of four drives that were terminated, and the number of hours they started acting strangely before they were replaced.

Fallacy *Computers systems achieve 99.999% availability ("five nines"), as advertised*

Marketing departments of companies making servers started bragging about the availability of their computer hardware; in terms of Figure D.21, they claim availability of 99.999%, nicknamed *five nines*. Even the marketing departments of operating system companies tried to give this impression.

Five minutes of unavailability per year is certainly impressive, but given the failure data collected in surveys, it's hard to believe. For example, Hewlett-Packard claims that the HP-9000 server hardware and HP-UX operating system can deliver

Messages in system log for failed disk	Number of log messages	Duration (hours)
Hardware Failure (Peripheral device write fault [for] Field Replaceable Unit)	1763	186
Not Ready (Diagnostic failure: ASCQ = Component ID [of] Field Replaceable Unit)	1460	90
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	1313	5
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	431	17

Figure D.20 Record in system log for 4 of the 368 disks in Tertiary Disk that were replaced over 18 months. See Talagala and Patterson [1999]. These messages, matching the SCSI specification, were placed into the system log by device drivers. Messages started occurring as much as a week before one drive was replaced by the operator. The third and fourth messages indicate that the drive's failure prediction mechanism detected and predicted imminent failure, yet it was still hours before the drives were replaced by the operator.

Unavailability (minutes per year)	**Availability (percent)**	**Availability class ("number of nines")**
50,000	90%	1
5000	99%	2
500	99.9%	3
50	99.99%	4
5	99.999%	5
0.5	99.9999%	6
0.05	99.99999%	7

Figure D.21 Minutes unavailable per year to achieve availability class. (From Gray and Siewiorek [1991].) Note that five nines mean unavailable five minutes per year.

a 99.999% availability guarantee "in certain pre-defined, pre-tested customer environments" (see Hewlett-Packard [1998]). This guarantee does not include failures due to operator faults, application faults, or environmental faults, which are likely the dominant fault categories today. Nor does it include scheduled downtime. It is also unclear what the financial penalty is to a company if a system does not match its guarantee.

Microsoft also promulgated a five nines marketing campaign. In January 2001, *www.microsoft.com* was unavailable for 22 hours. For its Web site to achieve 99.999% availability, it will require a clean slate for 250 years.

In contrast to marketing suggestions, well-managed servers typically achieve 99% to 99.9% availability.

Pitfall *Where a function is implemented affects its reliability*

In theory, it is fine to move the RAID function into software. In practice, it is very difficult to make it work reliably.

The software culture is generally based on eventual correctness via a series of releases and patches. It is also difficult to isolate from other layers of software. For example, proper software behavior is often based on having the proper version and patch release of the operating system. Thus, many customers have lost data due to software bugs or incompatibilities in environment in software RAID systems.

Obviously, hardware systems are not immune to bugs, but the hardware culture tends to place a greater emphasis on testing correctness in the initial release. In addition, the hardware is more likely to be independent of the version of the operating system.

Fallacy *Operating systems are the best place to schedule disk accesses*

Higher-level interfaces such as ATA and SCSI offer logical block addresses to the host operating system. Given this high-level abstraction, the best an OS can do is to try to sort the logical block addresses into increasing order. Since only the disk knows the mapping of the logical addresses onto the physical geometry of sectors, tracks, and surfaces, it can reduce the rotational and seek latencies.

For example, suppose the workload is four reads [Anderson 2003]:

Operation	Starting LBA	Length
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

The host might reorder the four reads into logical block order:

Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Depending on the relative location of the data on the disk, reordering could make it worse, as Figure D.22 shows. The disk-scheduled reads complete in three-quarters of a disk revolution, but the OS-scheduled reads take three revolutions.

Fallacy *The time of an average seek of a disk in a computer system is the time for a seek of one-third the number of cylinders*

This fallacy comes from confusing the way manufacturers market disks with the expected performance, and from the false assumption that seek times are linear in distance. The one-third-distance rule of thumb comes from calculating the distance of a seek from one random location to another random location, not including the current track and assuming there is a large number of tracks. In the past, manufacturers listed the seek of this distance to offer a consistent basis for comparison. (Today, they

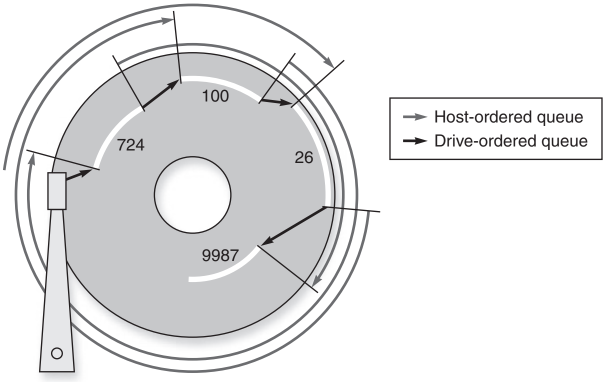


Figure D.22 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered. The former takes 3 revolutions to complete the 4 reads, while the latter completes them in just 3/4 of a revolution. (From Anderson [2003].)

calculate the “average” by timing all seeks and dividing by the number.) Assuming (incorrectly) that seek time is linear in distance, and using the manufacturer’s reported minimum and “average” seek times, a common technique to predict seek time is

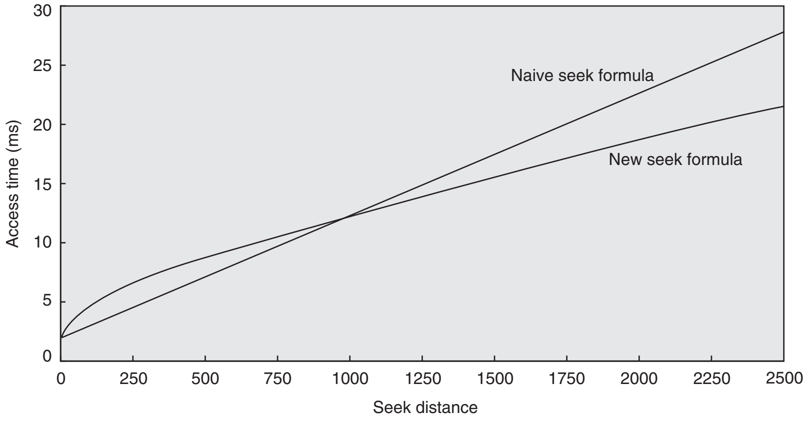
$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} \times \left(\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}}\right)$$

The fallacy concerning seek time is twofold. First, seek time is *not* linear with distance; the arm must accelerate to overcome inertia, reach its maximum traveling speed, decelerate as it reaches the requested position, and then wait to allow the arm to stop vibrating (*settle time*). Moreover, sometimes the arm must pause to control vibrations. For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as:

$$\text{Seek time}(\text{Distance}) = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

where a, b, and c are selected for a particular disk so that this formula will match the quoted times for Distance = 1, Distance = max, and Distance = 1/3 max. Figure D.23 plots this equation versus the fallacy equation. Unlike the first equation, the square root of the distance reflects acceleration and deceleration.

The second problem is that the average in the product specification would only be true if there were no locality to disk activity. Fortunately, there is both temporal and spatial locality (see page B-2 in Appendix B). For example, Figure D.24 shows sample measurements of seek distances for two workloads: a UNIX time-sharing workload and a business-processing workload. Notice the high percentage of disk



$$a = \frac{-10 \times \text{Time}_{\text{min}} + 15 \times \text{Time}_{\text{avg}} - 5 \times \text{Time}_{\text{max}}}{3 \times \sqrt{\text{Number of cylinders}}} \qquad b = \frac{7 \times \text{Time}_{\text{min}} - 15 \times \text{Time}_{\text{avg}} + 8 \times \text{Time}_{\text{max}}}{3 \times \text{Number of cylinders}} \qquad c = \text{Time}_{\text{min}}$$

Figure D.23 Seek time versus seek distance for sophisticated model versus naive model. Chen and Lee [1995] found that the equations shown above for parameters *a*, *b*, and *c* worked well for several disks.

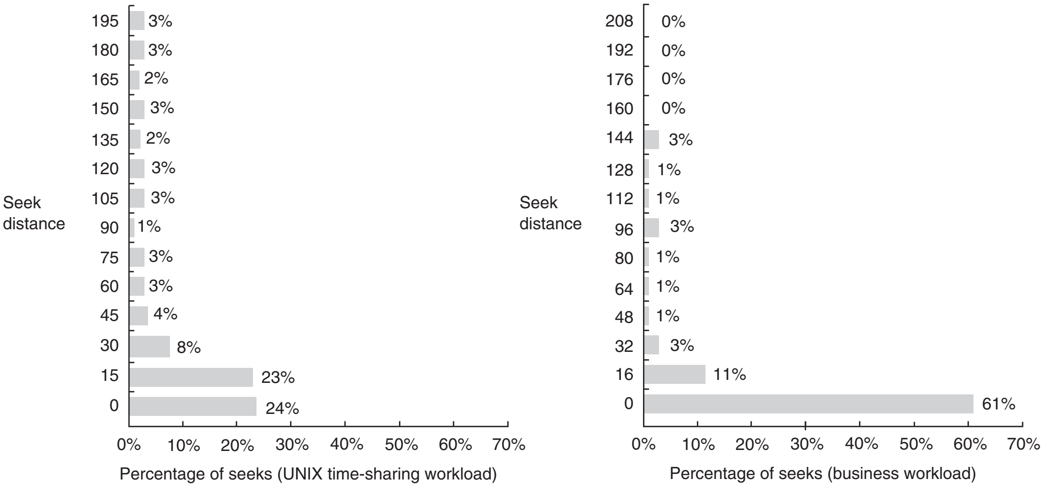


Figure D.24 Sample measurements of seek distances for two systems. The measurements on the left were taken on a UNIX time-sharing system. The measurements on the right were taken from a business-processing application in which the disk seek activity was scheduled to improve throughput. Seek distance of 0 means the access was made to the same cylinder. The rest of the numbers show the collective percentage for distances between numbers on the *y*-axis. For example, 11% for the bar labeled 16 in the business graph means that the percentage of seeks between 1 and 16 cylinders was 11%. The UNIX measurements stopped at 200 of the 1000 cylinders, but this captured 85% of the accesses. The business measurements tracked all 816 cylinders of the disks. The only seek distances with 1% or greater of the seeks that are not in the graph are 224 with 4%, and 304, 336, 512, and 624, each having 1%. This total is 94%, with the difference being small but nonzero distances in other categories. Measurements courtesy of Dave Anderson of Seagate.

accesses to the same cylinder, labeled distance 0 in the graphs, in both workloads. Thus, this fallacy couldn't be more misleading.

D.10 Concluding Remarks

Storage is one of those technologies that we tend to take for granted. And yet, if we look at the true status of things today, storage is king. One can even argue that servers, which have become commodities, are now becoming peripheral to storage devices. Driving that point home are some estimates from IBM, which expects storage sales to surpass server sales in the next two years.

Michael Vizard
Editor-in-chief, *Infoworld* (August 11, 2001)

As their value is becoming increasingly evident, storage systems have become the target of innovation and investment.

The challenges for storage systems today are dependability and maintainability. Not only do users want to be sure their data are never lost (reliability), applications today increasingly demand that the data are always available to access (availability). Despite improvements in hardware and software reliability and fault tolerance, the awkwardness of maintaining such systems is a problem both for cost and for availability. A widely mentioned statistic is that customers spend $6 to $8 operating a storage system for every $1 of purchase price. When dependability is attacked by having many redundant copies at a higher level of the system—such as for search—then very large systems can be sensitive to the price-performance of the storage components.

Today, challenges in storage dependability and maintainability dominate the challenges of I/O.

D.11 Historical Perspective and References

Section M.9 (available online) covers the development of storage devices and techniques, including who invented disks, the story behind RAID, and the history of operating systems and databases. References for further reading are included.

Case Studies with Exercises by Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

Case Study 1: Deconstructing a Disk

Concepts illustrated by this case study

- Performance Characteristics
- Microbenchmarks

The internals of a storage system tend to be hidden behind a simple interface, that of a linear array of blocks. There are many advantages to having a common interface for all storage systems: An operating system can use any storage system without modification, and yet the storage system is free to innovate behind this interface. For example, a single disk can map its internal < sector, track, surface > geometry to the linear array in whatever way achieves the best performance; similarly, a multidisk RAID system can map the blocks on any number of disks to this same linear array. However, this fixed interface has a number of disadvantages, as well; in particular, the operating system is not able to perform some performance, reliability, and security optimizations without knowing the precise layout of its blocks inside the underlying storage system.

In this case study, we will explore how software can be used to uncover the internal structure of a storage system hidden behind a block-based interface. The basic idea is to *fingerprint* the storage system: by running a well-defined workload on top of the storage system and measuring the amount of time required for different requests, one is able to infer a surprising amount of detail about the underlying system.

The Skippy algorithm, from work by Nisha Talagala and colleagues at the University of California–Berkeley, uncovers the parameters of a single disk. The key is to factor out disk rotational effects by making consecutive seeks to individual sectors with addresses that differ by a linearly increasing amount (increasing by 1, 2, 3, and so forth). Thus, the basic algorithm skips through the disk, increasing the distance of the seek by one sector before every write, and outputs the distance and time for each write. The raw device interface is used to avoid file system optimizations. The `SECTOR SIZE` is set equal to the minimum amount of data that can be read at once from the disk (e.g., 512 bytes). (Skippy is described in more detail in Talagala and Patterson [1999].)

```
fd = open("raw disk device");
for (i = 0; i < measurements; i++) {
    begin_time = gettime();
    lseek(fd, i*SECTOR_SIZE, SEEK_CUR);
    write(fd, buffer, SECTOR_SIZE);
    interval_time = gettime() -begin_time;

    printf("Stride: %d Time: %d\n", i, interval_time);
}
close(fd);
```

By graphing the time required for each write as a function of the seek distance, one can infer the minimal transfer time (with no seek or rotational latency), head switch time, cylinder switch time, rotational latency, and the number of heads in the disk. A typical graph will have four distinct lines, each with the same slope, but with different offsets. The highest and lowest lines correspond to requests that incur different amounts of rotational delay, but no cylinder or head switch costs; the difference between these two lines reveals the rotational latency of the disk. The second lowest line corresponds to requests that incur a head switch (in addition to increasing amounts of rotational delay). Finally, the third line corresponds to requests that incur a cylinder switch (in addition to rotational delay).

D.1 [10/10/10/10/10] <D.2> The results of running Skippy are shown for a mock disk (Disk Alpha) in Figure D.25.

a. [10] <D.2> What is the minimal transfer time?

b. [10] <D.2> What is the rotational latency?

c. [10] <D.2> What is the head switch time?

d. [10] <D.2> What is the cylinder switch time?

e. [10] <D.2> What is the number of disk heads?

D.2 [25] <D.2> Draw an approximation of the graph that would result from running Skippy on Disk Beta, a disk with the following parameters:

- Minimal transfer time, 2.0 ms
- Rotational latency, 6.0 ms

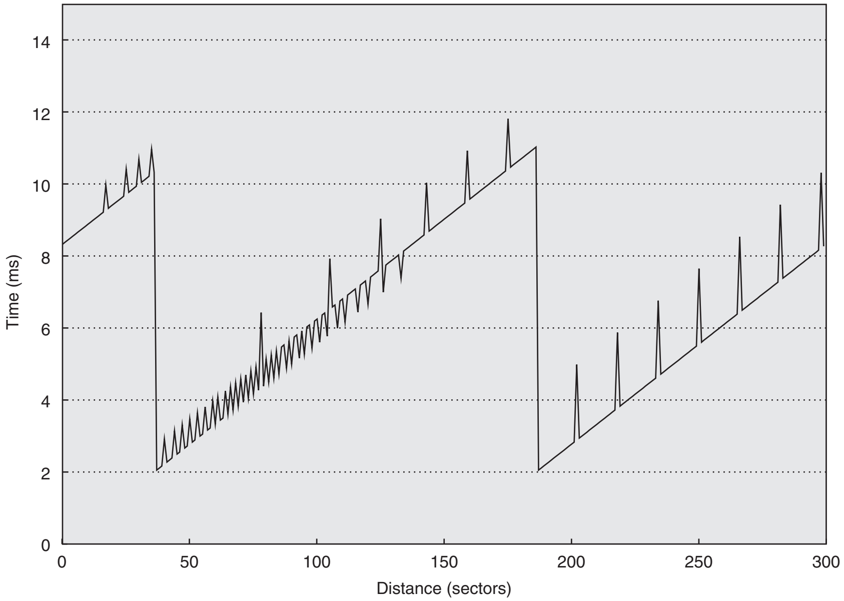


Figure D.25 Results from running Skippy on Disk Alpha.

- Head switch time, 1.0 ms
- Cylinder switch time, 1.5 ms
- Number of disk heads, 4
- Sectors per track, 100

D.3 [10/10/10/10/10/10/10] < D.2 > Implement and run the Skippy algorithm on a disk drive of your choosing.

a. [10] < D.2 > Graph the results of running Skippy. Report the manufacturer and model of your disk.

b. [10] < D.2 > What is the minimal transfer time?

c. [10] < D.2 > What is the rotational latency?

d. [10] < D.2 > What is the head switch time?

e. [10] < D.2 > What is the cylinder switch time?

f. [10] < D.2 > What is the number of disk heads?

g. [10] < D.2 > Do the results of running Skippy on a real disk differ in any qualitative way from that of the mock disk?

Case Study 2: Deconstructing a Disk Array

Concepts illustrated by this case study

- Performance Characteristics
- Microbenchmarks

The Shear algorithm, from work by Timothy Denehy and colleagues at the University of Wisconsin [Denehy et al. 2004], uncovers the parameters of a RAID system. The basic idea is to generate a workload of requests to the RAID array and time those requests; by observing which sets of requests take longer, one can infer which blocks are allocated to the same disk.

We define RAID properties as follows. Data are allocated to disks in the RAID at the block level, where a *block* is the minimal unit of data that the file system reads or writes from the storage system; thus, block size is known by the file system and the fingerprinting software. A *chunk* is a set of blocks that is allocated contiguously within a disk. A *stripe* is a set of chunks across each of D data disks. Finally, a *pattern* is the minimum sequence of data blocks such that block offset i within the pattern is always located on disk j.

D.4 [20/20] <D.2> One can uncover the pattern size with the following code. The code accesses the raw device to avoid file system optimizations. The key to all of the Shear algorithms is to use random requests to avoid triggering any of the prefetch or caching mechanisms within the RAID or within individual disks. The basic idea of this code sequence is to access N random blocks at a fixed interval p within the RAID array and to measure the completion time of each interval.

```
for (p = BLOCKSIZE; p <= testsize; p += BLOCKSIZE) {
    for (i = 0; i < N; i++) {
        request[i] = random()*p;
    }
     begin_time = gettime();

     issues all request[N] to raw device in parallel;

     wait for all request[N] to complete;
     interval_time = gettime() - begin_time;
     printf("PatternSize: %d Time: %d\n", p,
        interval_time);
}
```

If you run this code on a RAID array and plot the measured time for the N requests as a function of p, then you will see that the time is highest when all N requests fall on the same disk; thus, the value of p with the highest time corresponds to the pattern size of the RAID.

a. [20] <D.2> Figure D.26 shows the results of running the pattern size algorithm on an unknown RAID system.

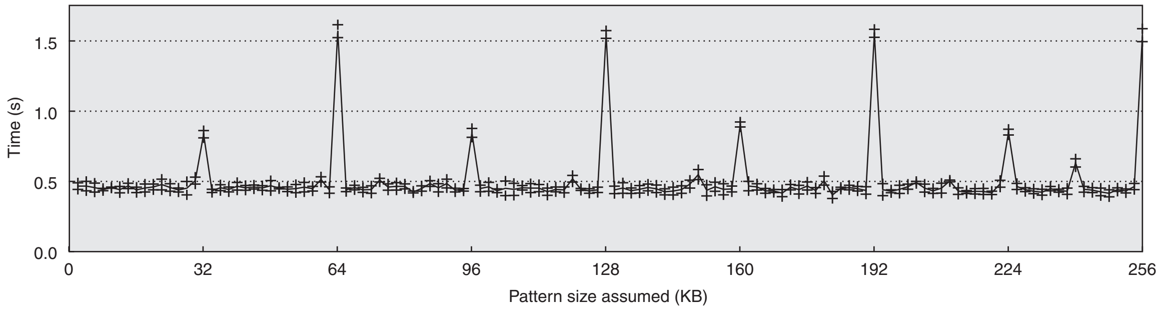


Figure D.26 **Results from running the pattern size algorithm of Shear on a mock storage system.**

- What is the pattern size of this storage system?
- What do the measured times of 0.4, 0.8, and 1.6 seconds correspond to in this storage system?
- If this is a RAID 0 array, then how many disks are present?
- If this is a RAID 0 array, then what is the chunk size?

b. [20] <D.2> Draw the graph that would result from running this Shear code on a storage system with the following characteristics:

- Number of requests, $N = 1000$
- Time for a random read on disk, 5 ms
- RAID level, RAID 0
- Number of disks, 4
- Chunk size, 8 KB

D.5 [20/20] <D.2> One can uncover the chunk size with the following code. The basic idea is to perform reads from N patterns chosen at random but always at controlled offsets, c and $c - 1$, within the pattern.

```
for (c = 0; c < patternsize; c += BLOCKSIZE) {
     for (i = 0; i < N; i++) {
          requestA[i] = random()*patternsize + c;
          requestB[i] = random()*patternsize +
               (c-1)%patternsize;
     }
          begin_time = gettime();

          issue all requestA[N] and requestB[N] to raw device
             in parallel;
          wait for requestA[N] and requestB[N] to complete;

          interval_time = gettime() - begin_time;
          printf("ChunkSize: %d Time: %d\n", c,
             interval_time);
   }
```

If you run this code and plot the measured time as a function of *c*, then you will see that the measured time is lowest when the *requestA* and *requestB* reads fall on two different disks. Thus, the values of *c* with low times correspond to the chunk boundaries between disks of the RAID.

a. [20] < D.2 > Figure D.27 shows the results of running the chunk size algorithm on an unknown RAID system.

- What is the chunk size of this storage system?
- What do the measured times of 0.75 and 1.5 seconds correspond to in this storage system?

b. [20] < D.2 > Draw the graph that would result from running this Shear code on a storage system with the following characteristics:

- Number of requests, $N = 1000$
- Time for a random read on disk, 5 ms
- RAID level, RAID 0
- Number of disks, 8
- Chunk size, 12 KB

D.6 [10/10/10/10] < D.2 > Finally, one can determine the layout of chunks to disks with the following code. The basic idea is to select *N* random patterns and to exhaustively read together all pairwise combinations of the chunks within the pattern.

```
for (a = 0; a < numchunks; a += chunksize) {
   for (b = a; b < numchunks; b += chunksize) {
      for (i = 0; i < N; i++) {
         requestA[i] = random()*patternsize + a;
         requestB[i] = random()*patternsize + b;
    }
    begin_time = gettime();
    issue all requestA[N] and requestB[N] to raw device
      in parallel;
```

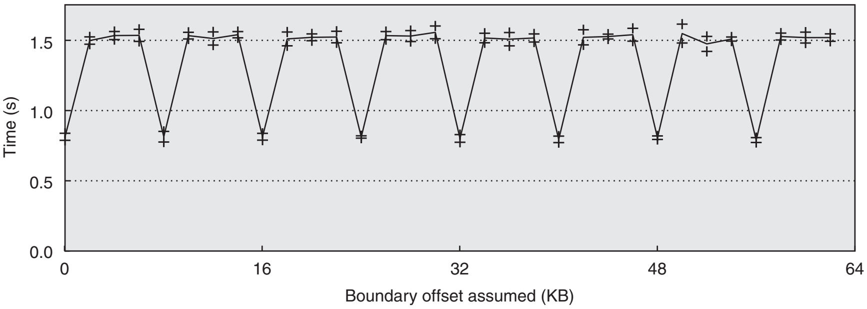


Figure D.27 Results from running the chunk size algorithm of Shear on a mock storage system.

```
            wait for all requestA[N] and requestB[N] to
              complete;
            interval_time = gettime() - begin_time;
            printf("A: %d B: %d Time: %d\n", a, b,
              interval_time);
          }
        }
```

After running this code, you can report the measured time as a function of *a* and *b*. The simplest way to graph this is to create a two-dimensional table with *a* and *b* as the parameters and the time scaled to a shaded value; we use darker shadings for faster times and lighter shadings for slower times. Thus, a light shading indicates that the two offsets of *a* and *b* within the pattern fall on the same disk.

Figure D.28 shows the results of running the layout algorithm on a storage system that is known to have a pattern size of 384 KB and a chunk size of 32 KB.

a. [20] <D.2> How many chunks are in a pattern?

b. [20] <D.2> Which chunks of each pattern appear to be allocated on the same disks?

c. [20] <D.2> How many disks appear to be in this storage system?

d. [20] <D.2> Draw the likely layout of blocks across the disks.

D.7 [20] <D.2> Draw the graph that would result from running the layout algorithm on the storage system shown in Figure D.29. This storage system has four disks and a chunk size of four 4 KB blocks (16 KB) and is using a RAID 5 Left-Asymmetric layout.

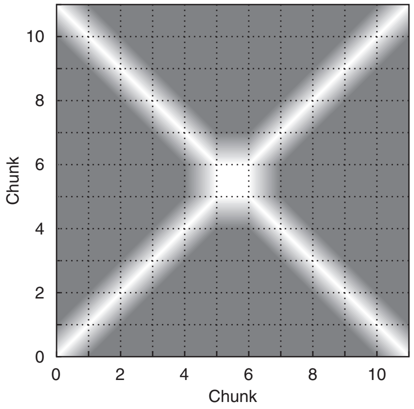
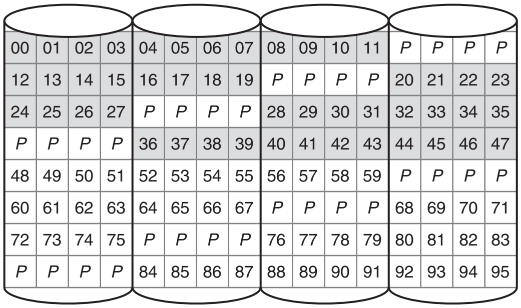


Figure D.28 Results from running the layout algorithm of Shear on a mock storage system.



00	01	02	03	04	05	06	07	08	09	10	11	*P*	*P*	*P*	*P*
12	13	14	15	16	17	18	19	*P*	*P*	*P*	*P*	20	21	22	23
24	25	26	27	*P*	*P*	*P*	*P*	28	29	30	31	32	33	34	35
P	*P*	*P*	*P*	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	*P*	*P*	*P*	*P*
60	61	62	63	64	65	66	67	*P*	*P*	*P*	*P*	68	69	70	71
72	73	74	75	*P*	*P*	*P*	*P*	76	77	78	79	80	81	82	83
P	*P*	*P*	*P*	84	85	86	87	88	89	90	91	92	93	94	95

Parity: RAID 5 Left-Asymmetric, stripe = 16, pattern = 48

Figure D.29 A storage system with four disks, a chunk size of four 4 KB blocks, and using a RAID 5 Left-Asymmetric layout. Two repetitions of the pattern are shown.

Case Study 3: RAID Reconstruction

Concepts illustrated by this case study

- RAID Systems
- RAID Reconstruction
- Mean Time to Failure (MTTF)
- Mean Time until Data Loss (MTDL)
- Performability
- Double Failures

A RAID system ensures that data are not lost when a disk fails. Thus, one of the key responsibilities of a RAID is to reconstruct the data that were on a disk when it failed; this process is called *reconstruction* and is what you will explore in this case study. You will consider both a RAID system that can tolerate one disk failure and a RAID-DP, which can tolerate two disk failures.

Reconstruction is commonly performed in two different ways. In *offline reconstruction*, the RAID devotes all of its resources to performing reconstruction and does not service any requests from the workload. In *online reconstruction*, the RAID continues to service workload requests while performing the reconstruction; the reconstruction process is often limited to use some fraction of the total bandwidth of the RAID system.

How reconstruction is performed impacts both the *reliability* and the *performability* of the system. In a RAID 5, data are lost if a second disk fails before the data from the first disk can be recovered; therefore, the longer the reconstruction time (MTTR), the lower the reliability or the *mean time until data loss* (MTDL). Performability is a metric meant to combine both the performance of a system and its availability; it is defined as the performance of the system in a given state multiplied by the probability of that state. For a RAID array, possible states include normal operation with no disk failures, reconstruction with one disk failure, and shutdown due to multiple disk failures.

For these exercises, assume that you have built a RAID system with six disks, plus a sufficient number of hot spares. Assume that each disk is the 37 GB SCSI disk shown in Figure D.3 and that each disk can sequentially read data at a peak of 142 MB/sec and sequentially write data at a peak of 85 MB/sec. Assume that the disks are connected to an Ultra320 SCSI bus that can transfer a total of 320 MB/sec. You can assume that each disk failure is independent and ignore other potential failures in the system. For the reconstruction process, you can assume that the overhead for any XOR computation or memory copying is negligible. During online reconstruction, assume that the reconstruction process is limited to use a total bandwidth of 10 MB/sec from the RAID system.

D.8 [10] <D.2> Assume that you have a RAID 4 system with six disks. Draw a simple diagram showing the layout of blocks across disks for this RAID system.

D.9 [10] <D.2, D.4> When a single disk fails, the RAID 4 system will perform reconstruction. What is the expected time until a reconstruction is needed?

D.10 [10/10/10] <D.2, D.4> Assume that reconstruction of the RAID 4 array begins at time *t*.

a. [10] <D.2, D.4> What read and write operations are required to perform the reconstruction?

b. [10] <D.2, D.4> For offline reconstruction, when will the reconstruction process be complete?

c. [10] <D.2, D.4> For online reconstruction, when will the reconstruction process be complete?

D.11 [10/10/10/10] <D.2, D.4> In this exercise, we will investigate the mean time until data loss (MTDL). In RAID 4, data are lost only if a second disk fails before the first failed disk is repaired.

a. [10] <D.2, D.4> What is the likelihood of having a second failure during offline reconstruction?

b. [10] <D.2, D.4> Given this likelihood of a second failure during reconstruction, what is the MTDL for offline reconstruction?

c. [10] <D.2, D.4> What is the likelihood of having a second failure during online reconstruction?

d. [10] <D.2, D.4> Given this likelihood of a second failure during reconstruction, what is the MTDL for online reconstruction?

D.12 [10] <D.2, D.4> What is performability for the RAID 4 array for offline reconstruction? Calculate the performability using IOPS, assuming a random readonly workload that is evenly distributed across the disks of the RAID 4 array.

D.13 [10] <D.2, D.4> What is the performability for the RAID 4 array for online reconstruction? During online repair, you can assume that the IOPS drop to 70% of their peak rate. Does offline or online reconstruction lead to better performability?

D.14 [10] <D.2, D.4> RAID 6 is used to tolerate up to two simultaneous disk failures. Assume that you have a RAID 6 system based on row-diagonal parity, or RAID-DP; your six-disk RAID-DP system is based on RAID 4, with $p = 5$, as shown in Figure D.5. If data disk 0 and data disk 3 fail, how can those disks be reconstructed? Show the sequence of steps that are required to compute the missing blocks in the first four stripes.

Case Study 4: Performance Prediction for RAIDs

Concepts illustrated by this case study

- RAID Levels
- Queuing Theory
- Impact of Workloads
- Impact of Disk Layout

In this case study, you will explore how simple queuing theory can be used to predict the performance of the I/O system. You will investigate how both storage system configuration and the workload influence service time, disk utilization, and average response time.

The configuration of the storage system has a large impact on performance. Different RAID levels can be modeled using queuing theory in different ways. For example, a RAID 0 array containing N disks can be modeled as N separate systems of M/M/1 queues, assuming that requests are appropriately distributed across the N disks. The behavior of a RAID 1 array depends upon the workload: A read operation can be sent to either mirror, whereas a write operation must be sent to both disks. Therefore, for a read-only workload, a two-disk RAID 1 array can be modeled as an M/M/2 queue, whereas for a write-only workload, it can be modeled as an M/M/1 queue. The behavior of a RAID 4 array containing N disks also depends upon the workload: A read will be sent to a particular data disk, whereas writes must all update the parity disk, which becomes the bottleneck of the system. Therefore, for a read-only workload, RAID 4 can be modeled as $N - 1$ separate systems, whereas for a write-only workload, it can be modeled as one M/M/1 queue.

The layout of blocks within the storage system can have a significant impact on performance. Consider a single disk with a 40 GB capacity. If the workload randomly accesses 40 GB of data, then the layout of those blocks to the disk does not have much of an impact on performance. However, if the workload randomly accesses only half of the disk's capacity (i.e., 20 GB of data on that disk), then layout does matter: To reduce seek time, the 20 GB of data can be compacted within 20 GB of consecutive tracks instead of allocated uniformly distributed over the entire 40 GB capacity.

For this problem, we will use a rather simplistic model to estimate the service time of a disk. In this basic model, the average positioning and transfer time for a small random request is a linear function of the seek distance. For the 40 GB disk in this problem, assume that the service time is 5 *ms* * space utilization. Thus, if the entire 40 GB disk is used, then the average positioning and transfer time for a random request is 5 ms; if only the first 20 GB of the disk is used, then the average positioning and transfer time is 2.5 ms.

Throughout this case study, you can assume that the processor sends 167 small random disk requests per second and that these requests are exponentially distributed. You can assume that the size of the requests is equal to the block size of 8 KB. Each disk in the system has a capacity of 40 GB. Regardless of the storage system configuration, the workload accesses a total of 40 GB of data; you should allocate the 40 GB of data across the disks in the system in the most efficient manner.

D.15 [10/10/10/10/10] <D.5> Begin by assuming that the storage system consists of a single 40 GB disk.

a. [10] <D.5> Given this workload and storage system, what is the average service time?

b. [10] <D.5> On average, what is the utilization of the disk?

c. [10] <D.5> On average, how much time does each request spend waiting for the disk?

d. [10] <D.5> What is the mean number of requests in the queue?

e. [10] <D.5> Finally, what is the average response time for the disk requests?

D.16 [10/10/10/10/10/10] <D.2, D.5> Imagine that the storage system is now configured to contain two 40 GB disks in a RAID 0 array; that is, the data are striped in blocks of 8 KB equally across the two disks with no redundancy.

a. [10] <D.2, D.5> How will the 40 GB of data be allocated across the disks? Given a random request workload over a total of 40 GB, what is the expected service time of each request?

b. [10] <D.2, D.5> How can queuing theory be used to model this storage system?

c. [10] <D.2, D.5> What is the average utilization of each disk?

d. [10] <D.2, D.5> On average, how much time does each request spend waiting for the disk?

e. [10] <D.2, D.5> What is the mean number of requests in each queue?

f. [10] <D.2, D.5> Finally, what is the average response time for the disk requests?

D.17 [20/20/20/20/20] <D.2, D.5> Instead imagine that the storage system is configured to contain two 40 GB disks in a RAID 1 array; that is, the data are mirrored across the two disks. Use queuing theory to model this system for a read-only workload.

a. [20] < D.2, D.5 > How will the 40 GB of data be allocated across the disks? Given a random request workload over a total of 40 GB, what is the expected service time of each request?

b. [20] < D.2, D.5 > How can queuing theory be used to model this storage system?

c. [20] < D.2, D.5 > What is the average utilization of each disk?

d. [20] < D.2, D.5 > On average, how much time does each request spend waiting for the disk?

e. [20] < D.2, D.5 > Finally, what is the average response time for the disk requests?

D.18 [10/10] < D.2, D.5 > Imagine that instead of a read-only workload, you now have a write-only workload on a RAID 1 array.

a. [10] < D.2, D.5 > Describe how you can use queuing theory to model this system and workload.

b. [10] < D.2, D.5 > Given this system and workload, what are the average utilization, average waiting time, and average response time?

Case Study 5: I/O Subsystem Design

Concepts illustrated by this case study

- RAID Systems
- Mean Time to Failure (MTTF)
- Performance and Reliability Trade-Offs

In this case study, you will design an I/O subsystem, given a monetary budget. Your system will have a minimum required capacity and you will optimize for performance, reliability, or both. You are free to use as many disks and controllers as fit within your budget.

Here are your building blocks:

- A 10,000 MIPS CPU costing $1000. Its MTTF is 1,000,000 hours.
- A 1000 MB/sec I/O bus with room for 20 Ultra320 SCSI buses and controllers.
- Ultra320 SCSI buses that can transfer 320 MB/sec and support up to 15 disks per bus (these are also called *SCSI strings*). The SCSI cable MTTF is 1,000,000 hours.
- An Ultra320 SCSI controller that is capable of 50,000 IOPS, costs $250, and has an MTTF of 500,000 hours.

- A $2000 enclosure supplying power and cooling to up to eight disks. The enclosure MTTF is 1,000,000 hours, the fan MTTF is 200,000 hours, and the power supply MTTF is 200,000 hours.
- The SCSI disks described in Figure D.3.
- Replacing any failed component requires 24 hours.

You may make the following assumptions about your workload:

- The operating system requires 70,000 CPU instructions for each disk I/O.
- The workload consists of many concurrent, random I/Os, with an average size of 16 KB.

All of your constructed systems must have the following properties:

- You have a monetary budget of $28,000.
- You must provide at least 1 TB of capacity.

D.19 [10] <D.2> You will begin by designing an I/O subsystem that is optimized only for capacity and performance (and not reliability), specifically IOPS. Discuss the RAID level and block size that will deliver the best performance.

D.20 [20/20/20/20] <D.2, D.4, D.7> What configuration of SCSI disks, controllers, and enclosures results in the best performance given your monetary and capacity constraints?

a. [20] <D.2, D.4, D.7> How many IOPS do you expect to deliver with your system?

b. [20] <D.2, D.4, D.7> How much does your system cost?

c. [20] <D.2, D.4, D.7> What is the capacity of your system?

d. [20] <D.2, D.4, D.7> What is the MTTF of your system?

D.21 [10] <D.2, D.4, D.7> You will now redesign your system to optimize for reliability, by creating a RAID 10 or RAID 01 array. Your storage system should be robust not only to disk failures but also to controller, cable, power supply, and fan failures as well; specifically, a single component failure should not prohibit accessing both replicas of a pair. Draw a diagram illustrating how blocks are allocated across disks in the RAID 10 and RAID 01 configurations. Is RAID 10 or RAID 01 more appropriate in this environment?

D.22 [20/20/20/20/20] <D.2, D.4, D.7> Optimizing your RAID 10 or RAID 01 array only for reliability (but staying within your capacity and monetary constraints), what is your RAID configuration?

a. [20] <D.2, D.4, D.7> What is the overall MTTF of the components in your system?

b. [20] < D.2, D.4, D.7 > What is the MTDL of your system?

c. [20] < D.2, D.4, D.7 > What is the usable capacity of this system?

d. [20] < D.2, D.4, D.7 > How much does your system cost?

e. [20] < D.2, D.4, D.7 > Assuming a write-only workload, how many IOPS can you expect to deliver?

D.23 [10] < D.2, D.4, D.7 > Assume that you now have access to a disk that has twice the capacity, for the same price. If you continue to design only for reliability, how would you change the configuration of your storage system? Why?

Case Study 6: Dirty Rotten Bits

Concepts illustrated by this case study

- Partial Disk Failure
- Failure Analysis
- Performance Analysis
- Parity Protection
- Checksumming

You are put in charge of avoiding the problem of "bit rot"—bits or blocks in a file going bad over time. This problem is particularly important in archival scenarios, where data are written once and perhaps accessed many years later; without taking extra measures to protect the data, the bits or blocks of a file may slowly change or become unavailable due to media errors or other I/O faults.

Dealing with bit rot requires two specific components: detection and recovery. To detect bit rot efficiently, one can use checksums over each block of the file in question; a checksum is just a function of some kind that takes a (potentially long) string of data as input and outputs a fixed-size string (the checksum) of the data as output. The property you will exploit is that if the data changes then the computed checksum is very likely to change as well.

Once detected, recovering from bit rot requires some form of redundancy. Examples include mirroring (keeping multiple copies of each block) and parity (some extra redundant information, usually more space efficient than mirroring).

In this case study, you will analyze how effective these techniques are given various scenarios. You will also write code to implement data integrity protection over a set of files.

D.24 [20/20/20] < D.2 > Assume that you will use simple parity protection in Exercises D.24 through D.27. Specifically, assume that you will be computing *one* parity block for each file in the file system. Further, assume that you will also use a 20-byte MD5 checksum per 4 KB block of each file.

We first tackle the problem of space overhead. According to studies by Douceur and Bolosky [1999], these file size distributions are what is found in modern PCs:

≤1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	≥1 MB
26.6%	11.0%	11.2%	10.9%	9.5%	8.5%	7.1%	5.1%	3.7%	2.4%	4.0%

The study also finds that file systems are usually about half full. Assume that you have a 37 GB disk volume that is roughly half full and follows that same distribution, and answer the following questions:

a. [20] < D.2 > How much extra information (both in bytes and as a percent of the volume) must you keep on disk to be able to detect a single error with checksums?

b. [20] < D.2 > How much extra information (both in bytes and as a percent of the volume) would you need to be able to both detect a single error with checksums as well as correct it?

c. [20] < D.2 > Given this file distribution, is the block size you are using to compute checksums too big, too little, or just right?

D.25 [10/10] < D.2, D.3 > One big problem that arises in data protection is error detection. One approach is to perform error detection *lazily*—that is, wait until a file is accessed, and at that point, check it and make sure the correct data are there. The problem with this approach is that files that are not accessed frequently may slowly rot away and when finally accessed have too many errors to be corrected. Hence, an eager approach is to perform what is sometimes called *disk scrubbing*—periodically go through all data and find errors proactively.

a. [10] < D.2, D.3 > Assume that bit flips occur independently, at a rate of 1 flip per GB of data per month. Assuming the same 20 GB volume that is half full, and assuming that you are using the SCSI disk as specified in Figure D.3 (4 ms seek, roughly 100 MB/sec transfer), how often should you scan through files to check and repair their integrity?

b. [10] < D.2, D.3 > At what bit flip rate does it become impossible to maintain data integrity? Again assume the 20 GB volume and the SCSI disk.

D.26 [10/10/10/10] < D.2, D.4 > Another potential cost of added data protection is found in performance overhead. We now study the performance overhead of this data protection approach.

a. [10] < D.2, D.4 > Assume we write a 40 MB file to the SCSI disk sequentially, and then write out the extra information to implement our data protection scheme to disk once. How much *write traffic* (both in total volume of bytes and as a percentage of total traffic) does our scheme generate?

b. [10] < D.2, D.4 > Assume we now are updating the file randomly, similar to a database table. That is, assume we perform a series of 4 KB random writes to the file, and each time we perform a single write, we must update the on-disk protection information. Assuming that we perform 10,000 random writes, how much *I/O traffic* (both in total volume of bytes and as a percentage of total traffic) does our scheme generate?

c. [10] < D.2, D.4 > Now assume that the data protection information is always kept in a separate portion of the disk, away from the file it is guarding (that is, assume for each file A, there is another file $A_{\text{checksums}}$ that holds all the check-sums for A). Hence, one potential overhead we must incur arises upon reads—that is, upon each read, we will use the checksum to detect data corruption.

   Assume you read 10,000 blocks of 4 KB each sequentially from disk. Assuming a 4 ms average seek cost and a 100 MB/sec transfer rate (like the SCSI disk in Figure D.3), how long will it take to read the file (and corresponding checksums) from disk? What is the time penalty due to adding checksums?

d. [10] < D.2, D.4 > Again assuming that the data protection information is kept separate as in part (c), now assume you have to read 10,000 random blocks of 4 KB each from a very large file (much bigger than 10,000 blocks, that is). For each read, you must again use the checksum to ensure data integrity. How long will it take to read the 10,000 blocks from disk, again assuming the same disk characteristics? What is the time penalty due to adding checksums?

D.27 [40] < D.2, D.3, D.4 > Finally, we put theory into practice by developing a user-level tool to guard against file corruption. Assume you are to write a simple set of tools to detect and repair data integrity. The first tool is used for checksums and parity. It should be called `build` and used like this:

```
build <filename>
```

The `build` program should then store the needed checksum and redundancy information for the file `filename` in a file in the same directory called `.filename.cp` (so it is easy to find later).

A second program is then used to check and potentially repair damaged files. It should be called `repair` and used like this:

```
repair <filename>
```

The `repair` program should consult the `.cp` file for the filename in question and verify that all the stored checksums match the computed checksums for the data. If the checksums don't match for a single block, `repair` should use the redundant information to reconstruct the correct data and fix the file. However, if two or more blocks are bad, `repair` should simply report that the file has been corrupted beyond repair. To test your system, we will provide a tool to corrupt files called `corrupt`. It works as follows:

```
corrupt <filename> <blocknumber>
```

All `corrupt` does is fill the specified block number of the file with random noise. For checksums you will be using MD5. MD5 takes an input string and gives you a

128-bit “fingerprint” or checksum as an output. A great and simple implementation of MD5 is available here:

http://sourceforge.net/project/showfiles.php?group_id=42360

Parity is computed with the XOR operator. In C code, you can compute the parity of two blocks, each of size `BLOCKSIZE`, as follows:

```
unsigned char block1[BLOCKSIZE];
unsigned char block2[BLOCKSIZE];

unsigned char parity[BLOCKSIZE];

// first, clear parity block
for (int i = 0; i < BLOCKSIZE; i++)
    parity[i] = 0;

// then compute parity; carat symbol does XOR in C
for (int i = 0; i < BLOCKSIZE; i++) {
    parity[i] = block1[i] ^block2[i];
}
```

Case Study 7: Sorting Things Out

Concepts illustrated by this case study

- Benchmarking
- Performance Analysis
- Cost/Performance Analysis
- Amortization of Overhead
- Balanced Systems

The database field has a long history of using benchmarks to compare systems. In this question, you will explore one of the benchmarks introduced by Anon. et al. [1985] (see Chapter 1): external, or disk-to-disk, sorting.

Sorting is an exciting benchmark for a number of reasons. First, sorting exercises a computer system across all its components, including disk, memory, and processors. Second, sorting at the highest possible performance requires a great deal of expertise about how the CPU caches, operating systems, and I/O subsystems work. Third, it is simple enough to be implemented by a student (see below!).

Depending on how much data you have, sorting can be done in one or multiple passes. Simply put, if you have enough memory to hold the entire dataset in memory, you can read the entire dataset into memory, sort it, and then write it out; this is called a “one-pass” sort.

If you do not have enough memory, you must sort the data in multiple passes. There are many different approaches possible. One simple approach is to sort each chunk of the input file and write it to disk; this leaves (input file size)/(memory size) sorted files on disk. Then, you have to merge each sorted temporary file into a final sorted output. This is called a "two-pass" sort. More passes are needed in the unlikely case that you cannot merge all the streams in the second pass.

In this case study, you will analyze various aspects of sorting, determining its effectiveness and cost-effectiveness in different scenarios. You will also write your own version of an external sort, measuring its performance on real hardware.

D.28 [20/20/20] < D.4 > We will start by configuring a system to complete a sort in the least possible time, with no limits on how much we can spend. To get peak bandwidth from the sort, we have to make sure all the paths through the system have sufficient bandwidth.

Assume for simplicity that the time to perform the in-memory sort of keys is linearly proportional to the CPU rate and memory bandwidth of the given machine (e.g., sorting 1 MB of records on a machine with 1 MB/sec of memory bandwidth and a 1 MIPS processor will take 1 second). Assume further that you have carefully written the I/O phases of the sort so as to achieve sequential bandwidth. And, of course, realize that if you don't have enough memory to hold all of the data at once that sort will take two passes.

One problem you may encounter in performing I/O is that systems often perform extra *memory copies*; for example, when the `read()` system call is invoked, data may first be read from disk into a system buffer and then subsequently copied into the specified user buffer. Hence, memory bandwidth during I/O can be an issue.

Finally, for simplicity, assume that there is no overlap of reading, sorting, or writing. That is, when you are reading data from disk, that is all you are doing; when sorting, you are just using the CPU and memory bandwidth; when writing, you are just writing data to disk.

Your job in this task is to configure a system to extract peak performance when sorting 1 GB of data (i.e., roughly 10 million 100-byte records). Use the following table to make choices about which machine, memory, I/O interconnect, and disks to buy.

CPU			**I/O interconnect**		
Slow	1 GIPS	$200	Slow	80 MB/sec	$50
Standard	2 GIPS	$1000	Standard	160 MB/sec	$100
Fast	4 GIPS	$2000	Fast	320 MB/sec	$400
Memory			**Disks**		
Slow	512 MB/sec	$100/GB	Slow	30 MB/sec	$70
Standard	1 GB/sec	$200/GB	Standard	60 MB/sec	$120
Fast	2 GB/sec	$500/GB	Fast	110 MB/sec	$300

Note: Assume that you are buying a single-processor system and that you can have up to two I/O interconnects. However, the amount of memory and number of disks are up to you (assume there is no limit on disks per I/O interconnect).

a. [20] < D.4 > What is the total cost of your machine? (Break this down by part, including the cost of the CPU, amount of memory, number of disks, and I/O bus.)

b. [20] < D.4 > How much time does it take to complete the sort of 1 GB worth of records? (Break this down into time spent doing reads from disk, writes to disk, and time spent sorting.)

c. [20] < D.4 > What is the bottleneck in your system?

D.29 [25/25/25] < D.4 > We will now examine cost-performance issues in sorting. After all, it is easy to buy a high-performing machine; it is much harder to buy a costeffective one.

One place where this issue arises is with the PennySort competition (*research.microsoft.com/barc/SortBenchmark/*). PennySort asks that you sort as many records as you can for a single penny. To compute this, you should assume that a system you buy will last for 3 years (94,608,000 seconds), and divide this by the total cost in pennies of the machine. The result is your time budget per penny.

Our task here will be a little simpler. Assume you have a fixed budget of $2000 (or less). What is the fastest sorting machine you can build? Use the same hardware table as in Exercise D.28 to configure the winning machine.

(*Hint:* You might want to write a little computer program to generate all the possible configurations.)

a. [25] < D.4 > What is the total cost of your machine? (Break this down by part, including the cost of the CPU, amount of memory, number of disks, and I/O bus.)

b. [25] < D.4 > How does the reading, writing, and sorting time break down with this configuration?

c. [25] < D.4 > What is the bottleneck in your system?

D.30 [20/20/20] < D.4, D.6 > Getting good disk performance often requires *amortization of overhead*. The idea is simple: If you must incur an overhead of some kind, do as much useful work as possible after paying the cost and hence reduce its impact. This idea is quite general and can be applied to many areas of computer systems; with disks, it arises with the seek and rotational costs (overheads) that you must incur before transferring data. You can amortize an expensive seek and rotation by transferring a large amount of data.

In this exercise, we focus on how to amortize seek and rotational costs during the second pass of a two-pass sort. Assume that when the second pass begins, there are *N* sorted runs on the disk, each of a size that fits within main memory. Our task here is to read in a chunk from each sorted run and merge the results into a final sorted output. Note that a read from one run will incur a seek and rotation, as it is very likely that the last read was from a different run.

a. [20] < D.4, D.6 > Assume that you have a disk that can transfer at 100 MB/sec, with an average seek cost of 7 ms, and a rotational rate of 10,000 RPM. Assume further that every time you read from a run, you read 1 MB of data and that there are 100 runs each of size 1 GB. Also assume that writes (to the final sorted output) take place in large 1 GB chunks. How long will the merge phase take, assuming I/O is the dominant (i.e., only) cost?

b. [20] < D.4, D.6 > Now assume that you change the read size from 1 MB to 10 MB. How is the total time to perform the second pass of the sort affected?

c. [20] < D.4, D.6 > In both cases, assume that what we wish to maximize is *disk efficiency*. We compute disk efficiency as the ratio of the time spent transferring data over the total time spent accessing the disk. What is the disk efficiency in each of the scenarios mentioned above?

D.31 [40] < D.2, D.4, D.6 > In this exercise, you will write your own external sort. To generate the data set, we provide a tool `generate` that works as follows:

```
generate <filename> <size (in MB)>
```

By running `generate`, you create a file named `filename` of size `size` MB. The file consists of 100 byte keys, with 10-byte records (the part that must be sorted).

We also provide a tool called `check` that checks whether a given input file is sorted or not. It is run as follows:

```
check <filename>
```

The basic one-pass sort does the following: reads in the data, sorts the data, and then writes the data out. However, numerous optimizations are available to you: overlapping reading and sorting, separating keys from the rest of the record for better cache behavior and hence faster sorting, overlapping sorting and writing, and so forth.

One important rule is that data must always start on disk (and not in the file system cache). The easiest way to ensure this is to unmount and remount the file system.

One goal: Beat the Datamation sort record. Currently, the record for sorting 1 million 100-byte records is 0.44 seconds, which was obtained on a cluster of 32 machines. If you are careful, you might be able to beat this on a single PC configured with a few disks.

E.1 Introduction E-2
E.2 Signal Processing and Embedded Applications: The Digital Signal Processor E-5
E.3 Embedded Benchmarks E-12
E.4 Embedded Multiprocessors E-14
E.5 Case Study: The Emotion Engine of the Sony PlayStation 2 E-15
E.6 Case Study: Sanyo VPC-SX500 Digital Camera E-19
E.7 Case Study: Inside a Cell Phone E-20
E.8 Concluding Remarks E-25

E

Embedded Systems

By Thomas M. Conte
North Carolina State University

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1 1/2 tons.

Popular Mechanics
March 1949

E.1 Introduction

Embedded computer systems—computers lodged in other devices where the presence of the computers is not immediately obvious—are the fastest-growing portion of the computer market. These devices range from everyday machines (most microwaves, most washing machines, printers, network switches, and automobiles contain simple to very advanced embedded microprocessors) to handheld digital devices (such as PDAs, cell phones, and music players) to video game consoles and digital set-top boxes. Although in some applications (such as PDAs) the computers are programmable, in many embedded applications the only programming occurs in connection with the initial loading of the application code or a later software upgrade of that application. Thus, the application is carefully tuned for the processor and system. This process sometimes includes limited use of assembly language in key loops, although time-to-market pressures and good software engineering practice restrict such assembly language coding to a fraction of the application.

Compared to desktop and server systems, embedded systems have a much wider range of processing power and cost—from systems containing low-end 8-bit and 16-bit processors that may cost less than a dollar, to those containing full 32-bit microprocessors capable of operating in the 500 MIPS range that cost approximately 10 dollars, to those containing high-end embedded processors that cost hundreds of dollars and can execute several billions of instructions per second. Although the range of computing power in the embedded systems market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price.

Embedded systems often process information in very different ways from general-purpose processors. Typically these applications include deadline-driven constraints—so-called *real-time constraints.* In these applications, a particular computation must be completed by a certain time or the system fails (there are other constraints considered real time, discussed in the next subsection).

Embedded systems applications typically involve processing information as *signals.* The lay term “signal” often connotes radio transmission, and that is true for some embedded systems (e.g., cell phones). But a signal may be an image, a motion picture composed of a series of images, a control sensor measurement, and so on. Signal processing requires specific computation that many embedded processors are optimized for. We discuss this in depth below. A wide range of benchmark requirements exist, from the ability to run small, limited code segments to the ability to perform well on applications involving tens to hundreds of thousands of lines of code.

Two other key characteristics exist in many embedded applications: the need to minimize memory and the need to minimize power. In many embedded applications, the memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. Sometimes the application is expected to fit entirely in the memory on the processor chip; other times the application needs to fit in its entirety in a small, off-chip memory. In either case, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application. Some architectures have special instruction set capabilities to reduce code size. Larger memories also mean more power, and optimizing power is often critical in embedded applications. Although the emphasis on low power is frequently driven by the use of batteries, the need to use less expensive packaging (plastic versus ceramic) and the absence of a fan for cooling also limit total power consumption. We examine the issue of power in more detail later in this appendix.

Another important trend in embedded systems is the use of processor cores together with application-specific circuitry—so-called "core plus ASIC" or "system on a chip" (SOC), which may also be viewed as special-purpose multiprocessors (see Section E.4). Often an application's functional and performance requirements are met by combining a custom hardware solution together with software running on a standardized embedded processor core, which is designed to interface to such special-purpose hardware. In practice, embedded problems are usually solved by one of three approaches:

1. The designer uses a combined hardware/software solution that includes some custom hardware and an embedded processor core that is integrated with the custom hardware, often on the same chip.
2. The designer uses custom software running on an off-the-shelf embedded processor.
3. The designer uses a digital signal processor and custom software for the processor. *Digital signal processors* are processors specially tailored for signal-processing applications. We discuss some of the important differences between digital signal processors and general-purpose embedded processors below.

Figure E.1 summarizes these three classes of computing environments and their important characteristics.

Real-Time Processing

Often, the performance requirement in an embedded application is a real-time requirement. A *real-time performance requirement* is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the frame before the next frame arrives (typically called *hard real-time systems*). In some applications, a more sophisticated requirement exists: The average time for a particular task is constrained as well as is the number of instances when some maximum time is exceeded. Such approaches (typically called *soft real-time*) arise when it is possible to occasionally miss the time constraint on an event, as long as not too many are missed. Real-time

Feature	**Desktop**	**Server**	**Embedded**
Price of system	$1000–$10,000	$10,000–$10,000,000	$10–$100,000 (including network routers at the high end)
Price of microprocessor module	$100–$1000	$200–$2000 (per processor)	$0.20–$200 (per processor)
Microprocessors sold per year (estimates for 2000)	150,000,000	4,000,000	300,000,000 (32-bit and 64-bit processors only)
Critical system design issues	Price-performance, graphics performance	Throughput, availability, scalability	Price, power consumption, application-specific performance

Figure E.1 A summary of the three computing classes and their system characteristics. Note the wide range in system price for servers and embedded systems. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing and Web server applications. For embedded systems, one significant high-end application is a network router, which could include multiple processors as well as lots of memory and other electronics. The total number of embedded processors sold in 2000 is estimated to exceed 1 billion, if you include 8-bit and 16-bit microprocessors. In fact, the largest-selling microprocessor of all time is an 8-bit microcontroller sold by Intel! It is difficult to separate the low end of the server market from the desktop market, since low-end servers—especially those costing less than $5000—are essentially no different from desktop PCs. Hence, up to a few million of the PC units may be effectively servers.

performance tends to be highly application dependent. It is usually measured using kernels either from the application or from a standardized benchmark (see Section E.3).

The construction of a hard real-time system involves three key variables. The first is the rate at which a particular task must occur. Coupled to this are the hardware and software required to achieve that real-time rate. Often, structures that are very advantageous on the desktop are the enemy of hard real-time analysis. For example, branch speculation, cache memories, and so on introduce *uncertainty* into code. A particular sequence of code may execute either very efficiently or very inefficiently, depending on whether the hardware branch predictors and caches "do their jobs." Engineers must analyze code assuming the *worst-case execution time* (WCET). In the case of traditional microprocessor hardware, if one assumes that *all branches are mispredicted* and *all caches miss,* the WCET is overly pessimistic. Thus, the system designer may end up overdesigning a system to achieve a given WCET, when a much less expensive system would have sufficed.

In order to address the challenges of hard real-time systems, and yet still exploit such well-known architectural properties as branch behavior and access locality, it is possible to change how a processor is designed. Consider branch prediction: Although dynamic branch prediction is known to perform far more accurately than static "hint bits" added to branch instructions, the behavior of static hints is much more predictable. Furthermore, although caches perform better than software-managed on-chip memories, the latter produces predictable memory latencies. In some embedded processors, caches can be converted into software-managed on-chip memories via *line locking.* In this approach, a cache line can be locked in the cache so that it cannot be replaced until the line is unlocked

E.2 Signal Processing and Embedded Applications: The Digital Signal Processor

A digital signal processor (DSP) is a special-purpose processor optimized for executing digital signal processing algorithms. Most of these algorithms, from time-domain filtering (e.g., infinite impulse response and finite impulse response filtering), to convolution, to transforms (e.g., fast Fourier transform, discrete cosine transform), to even forward error correction (FEC) encodings, all have as their kernel the same operation: a multiply-accumulate operation. For example, the discrete Fourier transform has the form:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \text{ where } W_N^{kn} = e^{j\frac{2\pi kn}{N}} = \cos\left(2\pi \frac{kn}{N}\right) + j\sin\left(2\pi \frac{kn}{N}\right)$$

The discrete cosine transform is often a replacement for this because it does not require complex number operations. Either transform has as its core the *sum* of a *product*. To accelerate this, DSPs typically feature special-purpose hardware to perform *multiply-accumulate* (MAC). A MAC instruction of "MAC A,B,C" has the semantics of "A = A + B * C." In some situations, the performance of this operation is so critical that a DSP is selected for an application based solely upon its MAC operation throughput.

DSPs often employ *fixed-point* arithmetic. If you think of integers as having a binary point to the right of the least-significant bit, fixed point has a binary point just to the right of the sign bit. Hence, fixed-point data are fractions between −1 and +1.

Example Here are three simple 16-bit patterns:

0100 0000 0000 0000

0000 1000 0000 0000

0100 1000 0000 1000

What values do they represent if they are two's complement integers? Fixedpoint numbers?

Answer Number representation tells us that the *i*th digit to the left of the binary point represents 2^{i-1} and the *i*th digit to the right of the binary point represents 2^{-i}. First assume these three patterns are integers. Then the binary point is to the far right, so they represent 2^{14}, 2^{11}, and $(2^{14}+ 2^{11}+ 2^{3})$, or 16,384, 2048, and 18,440.

Fixed point places the binary point just to the right of the sign bit, so as fixed point these patterns represent 2^{-1}, 2^{-4}, and $(2^{-1} + 2^{-4} + 2^{-12})$. The fractions are 1/2, 1/16, and (2048 + 256 + 1)/4096 or 2305/4096, which represents about 0.50000, 0.06250, and 0.56274. Alternatively, for an *n*-bit two's complement, fixed-point number we could just divide the integer presentation by 2^{n-1} to derive the same results:

$$16{,}384/32{,}768 = 1/2,\ 2048/32{,}768 = 1/16,\ \text{and}\ 18{,}440/32{,}768 = 2305/4096.$$

Fixed point can be thought of as a low-cost floating point. It doesn't include an exponent in every word and doesn't have hardware that automatically aligns and normalizes operands. Instead, fixed point relies on the DSP programmer to keep the exponent in a separate variable and ensure that each result is shifted left or right to keep the answer aligned to that variable. Since this exponent variable is often shared by a set of fixed-point variables, this style of arithmetic is also called *blocked floating point*, since a block of variables has a common exponent.

To support such manual calculations, DSPs usually have some registers that are wider to guard against round-off error, just as floating-point units internally have extra guard bits. Figure E.2 surveys four generations of DSPs, listing data sizes and width of the accumulating registers. Note that DSP architects are not bound by the powers of 2 for word sizes. Figure E.3 shows the size of data operands for the TI TMS320C55 DSP.

In addition to MAC operations, DSPs often also have operations to accelerate portions of communications algorithms. An important class of these algorithms revolve around encoding and decoding *forward error correction codes*—codes in which extra information is added to the digital bit stream to guard against errors in transmission. A code of rate m/n has m information bits for $(m + n)$ check bits. So, for example, a 1/2 rate code would have 1 information bit per every 2 bits. Such codes are often called *trellis codes* because one popular graphical flow diagram of

Generation	Year	Example DSP	Data width	Accumulator width
1	1982	TI TMS32010	16 bits	32 bits
2	1987	Motorola DSP56001	24 bits	56 bits
3	1995	Motorola DSP56301	24 bits	56 bits
4	1998	TI TMS320C6201	16 bits	40 bits

Figure E.2 Four generations of DSPs, their data width, and the width of the registers that reduces round-off error.

Data size	Memory operand in operation	Memory operand in data transfer
16 bits	89.3%	89.0%
32 bits	10.7%	11.0%

Figure E.3 Size of data operands for the TMS320C55 DSP. About 90% of operands are 16 bits. This DSP has two 40-bit accumulators. There are no floating-point operations, as is typical of many DSPs, so these data are all fixed-point integers.

their encoding resembles a garden trellis. A common algorithm for decoding trellis codes is due to Viterbi. This algorithm requires a sequence of compares and selects in order to recover a transmitted bit's true value. Thus DSPs often have compare-select operations to support Viterbi decode for FEC codes.

To explain DSPs better, we will take a detailed look at two DSPs, both produced by Texas Instruments. The TMS320C55 series is a DSP family targeted toward battery-powered embedded applications. In stark contrast to this, the TMS VelociTI 320C6x series is a line of powerful, eight-issue VLIW processors targeted toward a broader range of applications that may be less power sensitive.

The TI 320C55

At one end of the DSP spectrum is the TI 320C55 architecture. The C55 is optimized for low-power, embedded applications. Its overall architecture is shown in Figure E.4. At the heart of it, the C55 is a seven-staged pipelined CPU. The stages are outlined below:

- *Fetch stage* reads program data from memory into the instruction buffer queue.
- *Decode stage* decodes instructions and dispatches tasks to the other primary functional units.
- *Address stage* computes addresses for data accesses and branch addresses for program discontinuities.
- *Access 1/Access 2 stages* send data read addresses to memory.
- *Read stage* transfers operand data on the B bus, C bus, and D bus.
- *Execute stage* executes operation in the A unit and D unit and performs writes on the E bus and F bus.

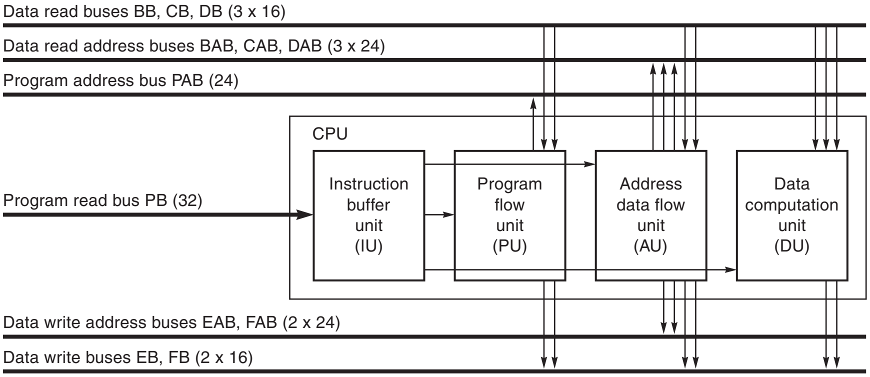

Data read buses BB, CB, DB (3 x 16)
Data read address buses BAB, CAB, DAB (3 x 24)
Program address bus PAB (24)
CPU
Program read bus PB (32)
Instruction buffer unit (IU)
Program flow unit (PU)
Address data flow unit (AU)
Data computation unit (DU)
Data write address buses EAB, FAB (2 x 24)
Data write buses EB, FB (2 x 16)

Figure E.4 Architecture of the TMS320C55 DSP. The C55 is a seven-stage pipelined processor with some unique instruction execution facilities. (Courtesy Texas Instruments.)

The C55 pipeline performs pipeline hazard detection and will stall on write after read (WAR) and read after write (RAW) hazards.

The C55 does have a 24 KB instruction cache, but it is configurable to support various workloads. It may be configured to be two-way set associative, direct-mapped, or as a "ramset." This latter mode is a way to support hard realtime applications. In this mode, blocks in the cache cannot be replaced.

The C55 also has advanced power management. It allows dynamic power management through software-programmable "idle domains." Blocks of circuitry on the device are organized into these idle domains. Each domain can operate normally or can be placed in a low-power idle state. A programmer-accessible Idle Control Register (ICR) determines which domains will be placed in the idle state when the execution of the next `IDLE` instruction occurs. The six domains are CPU, direct memory access (DMA), peripherals, clock generator, instruction cache, and external memory interface. When each domain is in the idle state, the functions of that particular domain are not available. However, in the peripheral domain, each peripheral has an Idle Enable bit that controls whether or not the peripheral will respond to the changes in the idle state. Thus, peripherals can be individually configured to idle or remain active when the peripheral domain is idled.

Since the C55 is a DSP, the central feature is its MAC units. The C55 has two MAC units, each comprised of a 17-bit by 17-bit multiplier coupled to a 40-bit dedicated adder. Each MAC unit performs its work in a single cycle; thus, the C55 can execute two MACs per cycle in full pipelined operation. This kind of capability is critical for efficiently performing signal processing applications. The C55 also has a compare, select, and store unit (CSSU) for the add/compare section of the Viterbi decoder.

The TI 320C6x

In stark contrast to the C55 DSP family is the high-end Texas Instruments VelociTI 320C6x family of processors. The C6x processors are closer to traditional very long instruction word (VLIW) processors because they seek to exploit the high levels of instruction-level parallelism (ILP) in many signal processing algorithms. Texas Instruments is not alone in selecting VLIW for exploiting ILP in the embedded space. Other VLIW DSP vendors include Ceva, StarCore, Philips/TriMedia, and STMicroelectronics. Why do these vendors favor VLIW over superscalar? For the embedded space, code compatibility is less of a problem, and so new applications can be either hand tuned or recompiled for the newest generation of processor. The other reason superscalar excels on the desktop is because the compiler cannot predict memory latencies at compile time. In embedded, however, memory latencies are often much more predictable. In fact, hard real-time constraints force memory latencies to be statically predictable. Of course, a superscalar would also perform well in this environment with these constraints, but the extra hardware to dynamically schedule instructions is both wasteful in terms of precious chip area and in terms of power consumption. Thus VLIW is a natural choice for high-performance embedded.

The C6x family employs different pipeline depths depending on the family member. For the C64x, for example, the pipeline has 11 stages. The first four stages of the pipeline perform instruction fetch, followed by two stages for instruction decode, and finally four stages for instruction execution. The overall architecture of the C64x is shown below in Figure E.5.

The C6x family's execution stage is divided into two parts, the left or "1" side and the right or "2" side. The L1 and L2 units perform logical and arithmetic operations. D units in contrast perform a subset of logical and arithmetic operations but also perform memory accesses (loads and stores). The two M units perform multiplication and related operations (e.g., shifts). Finally the S units perform comparisons, branches, and some SIMD operations (see the next subsection for a detailed explanation of SIMD operations). Each side has its own 32-entry, 32-bit register file (the A file for the 1 side, the B file for the 2 side). A side may access the other side's registers, but with a 1- cycle penalty. Thus, an instruction executing on side 1 may access B5, for example, but it will take 1- cycle extra to execute because of this.

VLIWs are traditionally very bad when it comes to code size, which runs contrary to the needs of embedded systems. However, the C6x family's approach "compresses" instructions, allowing the VLIW code to achieve the same density as equivalent RISC (reduced instruction set computer) code. To do so, instruction fetch is carried out on an "instruction packet," shown in Figure E.6. Each instruction has a *p* bit that specifies whether this instruction is a member of the current

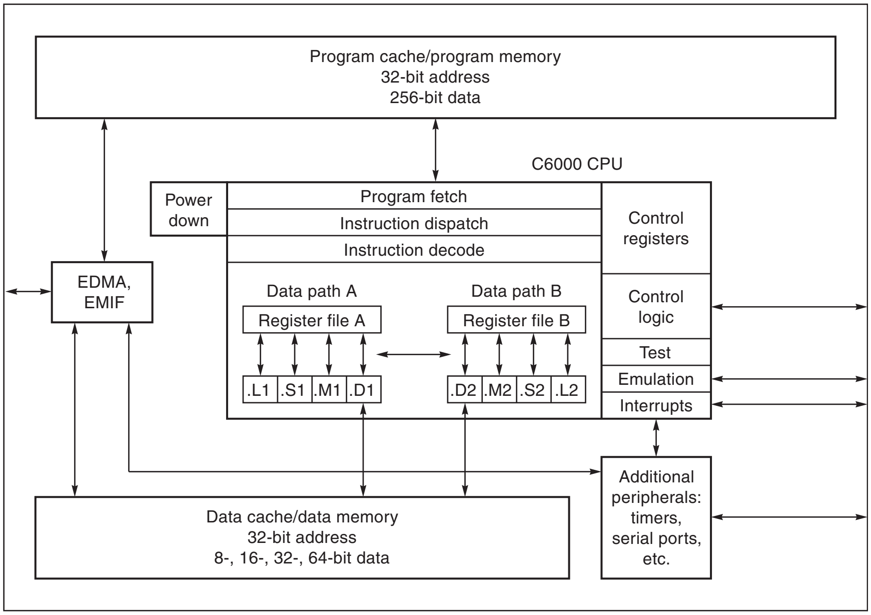

Program cache/program memory
32-bit address
256-bit data
C6000 CPU
Power down
Program fetch
Instruction dispatch
Instruction decode
Control registers
EDMA, EMIF
Data path A
Register file A
.L1
.S1
.M1
.D1
Data path B
Register file B
.D2
.M2
.S2
.L2
Control logic
Test
Emulation
Interrupts
Additional peripherals: timers, serial ports, etc.
Data cache/data memory
32-bit address
8-, 16-, 32-, 64-bit data

Figure E.5 Architecture of the TMS320C64x family of DSPs. The C6x is an eight-issue traditional VLIW processor. (Courtesy Texas Instruments.)

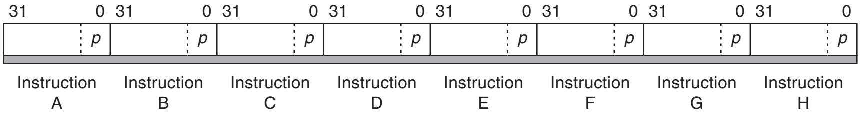


Figure E.6 Instruction packet of the TMS320C6x family of DSPs. The *p* bits determine whether an instruction begins a new VLIW word or not. If the *p* bit of instruction *i* is 1, then instruction *i* + 1 is to be executed in parallel with (in the same cycle as) instruction *i*. If the *p* bit of instruction *i* is 0, then instruction *i* + 1 is executed in the cycle after instruction *i*. (Courtesy Texas Instruments.)

VLIW word or the next VLIW word (see the figure for a detailed explanation). Thus, there are now no NOPs that are needed for VLIW encoding.

Software pipelining is an important technique for achieving high performance in a VLIW. But software pipelining relies on each iteration of the loop having an identical schedule to all other iterations. Because conditional branch instructions disrupt this pattern, the C6x family provides a means to conditionally execute instructions using *predication.* In predication, the instruction performs its work. But when it is done executing, an additional register, for example A1, is checked. If A1 is zero, the instruction does not write its results. If A1 is nonzero, the instruction proceeds normally. This allows simple if-then and if-then-else structures to be collapsed into straight-line code for software pipelining.

Media Extensions

There is a middle ground between DSPs and microcontrollers: *media extensions.* These extensions add DSP-like capabilities to microcontroller architectures at relatively low cost. Because media processing is judged by human perception, the data for multimedia operations are often much narrower than the 64-bit data word of modern desktop and server processors. For example, floating-point operations for graphics are normally in single precision, not double precision, and often at a precision less than is required by IEEE 754. Rather than waste the 64-bit arithmetic-logical units (ALUs) when operating on 32-bit, 16-bit, or even 8-bit integers, multimedia instructions can operate on several narrower data items at the same time. Thus, a *partitioned add* operation on 16-bit data with a 64-bit ALU would perform four 16-bit adds in a single clock cycle. The extra hardware cost is simply to prevent carries between the four 16-bit partitions of the ALU. For example, such instructions might be used for graphical operations on pixels. These operations are commonly called *single-instruction multiple-data* (SIMD) or *vector* instructions.

Most graphics multimedia applications use 32-bit floating-point operations. Some computers double peak performance of single-precision, floating-point operations; they allow a single instruction to launch two 32-bit operations on operands found side by side in a double-precision register. The two partitions must be insulated to prevent operations on one half from affecting the other. Such floating-point operations are called *paired single operations.* For example, such an operation might be used for graphical transformations of vertices. This doubling in performance is typically accomplished by doubling the number of floating-point units, making it more expensive than just suppressing carries in integer adders.

Figure E.7 summarizes the SIMD multimedia instructions found in several recent computers.

DSPs also provide operations found in the first three rows of Figure E.7, but they change the semantics a bit. First, because they are often used in real-time applications, there is not an option of causing an exception on arithmetic overflow (otherwise it could miss an event); thus, the result will be used no matter what the inputs. To support such an unyielding environment, DSP architectures use *saturating arithmetic*: If the result is too large to be represented, it is set to the largest representable number, depending on the sign of the result. In contrast, two's complement arithmetic can add a small positive number to a large positive.

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	PowerPC AltiVec	SPARC VIS
Add/subtract		4H	8B, 4H, 2W	16B, 8H, 4W	4H, 2W
Saturating add/subtract		4H	8B, 4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B (>=)		8B, 4H, 2W (=, >)	16B, 8H, 4W (=, >, > =, <, <=)	4H, 2W (=, not =, >, <=)
Shift right/left		4H	4H, 2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
AND/OR/XOR	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W	16B, 8H, 4W	8B, 4H, 2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack (2*n* bits → *n* bits)	2W → 2B, 4H → 4B	2*4H → 8B	4H → 4B, 2W → 2H	4W → 4B, 8H → 8B	2W → 2H, 2W → 2B, 4H → 4B
Unpack/merge	2B → 2W, 4B → 4H		2B → 2W, 4B → 4H	4B → 4W, 8B → 8H	4B → 4H, 2*4B → 8B
Permute/shuffle		4H		16B, 8H, 4W	

Figure E.7 Summary of multimedia support for desktop processors. Note the diversity of support, with little in common across the five architectures. All are fixed-width operations, performing multiple narrow operations on either a 64-bit or 128-bit ALU. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that AltiVec assumes a 128-bit ALU, and the rest assume 64 bits. Pack and unpack use the notation 2*2W to mean 2 operands each with 2 words. This table is a simplification of the full multimedia architectures, leaving out many details. For example, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of AltiVec, MAX, and VIS.

E.3 Embedded Benchmarks

It used to be the case just a couple of years ago that in the embedded market, many manufacturers quoted Dhrystone performance, a benchmark that was criticized and given up by desktop systems more than 20 years ago! As mentioned earlier, the enormous variety in embedded applications, as well as differences in performance requirements (hard real time, soft real time, and overall cost-performance), make the use of a single set of benchmarks unrealistic. In practice, many designers of embedded systems devise benchmarks that reflect their application, either as kernels or as stand-alone versions of the entire application.

For those embedded applications that can be characterized well by kernel performance, the best standardized set of benchmarks appears to be a new benchmark set: the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced "embassy"). The EEMBC benchmarks fall into six classes (called "subcommittees" in the parlance of EEMBC): automotive/industrial, consumer, telecommunications, digital entertainment, networking (currently in its second version), and office automation (also the second version of this subcommittee). Figure E.8 shows the six different application classes, which include 50 benchmarks.

Although many embedded applications are sensitive to the performance of small kernels, remember that often the overall performance of the entire application (which may be thousands of lines) is also critical. Thus, for many embedded systems, the EMBCC benchmarks can only be used to partially assess performance.

Benchmark type ("subcommittee")	**Number of kernels**	**Example benchmarks**
Automotive/industrial	16	6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or FFT benchmarks
Consumer	5	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions)
Telecommunications	5	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, encoder)
Digital entertainment	12	MP3 decode, MPEG-2 and MPEG-4 encode and decode (each of which is applied to five different datasets), MPEG Encode Floating Point, 4 benchmark tests for common cryptographic standards and algorithms (AES, DES, RSA, and Huffman decoding for data decompression), and enhanced JPEG and color-space conversion tests
Networking version 2	6	IP Packet Check (borrowed from the RFC1812 standard), IP Reassembly, IP Network Address Translator (NAT), Route Lookup, OSPF, Quality of Service (QOS), and TCP
Office automation version 2	6	Ghostscript, text parsing, image rotation, dithering, Bézier

Figure E.8 The EEMBC benchmark suite, consisting of 50 kernels in six different classes. See *www.eembc.org* for more information on the benchmarks and for scores.

Power Consumption and Efficiency as the Metric

Cost and power are often at least as important as performance in the embedded market. In addition to the cost of the processor module (which includes any required interface chips), memory is often the next most costly part of an embedded system. Unlike a desktop or server system, most embedded systems do not have secondary storage; instead, the entire application must reside in either FLASH or DRAM. Because many embedded systems, such as PDAs and cell phones, are constrained by both cost and physical size, the amount of memory needed for the application is critical. Likewise, power is often a determining factor in choosing a processor, especially for battery-powered systems.

EEMBC EnergyBench provides data on the amount of energy a processor consumes while running EEMBC's performance benchmarks. An EEMBC-certified Energymark score is an optional metric that a device manufacturer may choose to supply in conjunction with certified scores for device performance as a way of indicating a processor's efficient use of power and energy. EEMBC has standardized on the use of National Instruments' LabVIEW graphical development environment and data acquisition hardware to implement EnergyBench.

Figure E.9 shows the relative performance per watt of typical operating power. Compare this figure to Figure E.10, which plots raw performance, and notice how different the results are. The NEC VR 4122 has a clear advantage in performance per watt, but is the second-lowest performing processor! From the viewpoint of power consumption, the NEC VR 4122, which was designed for battery-based systems, is the big winner. The IBM PowerPC displays efficient use of power to achieve its high performance, although at 6 W typical, it is probably not suitable for most battery-based devices.

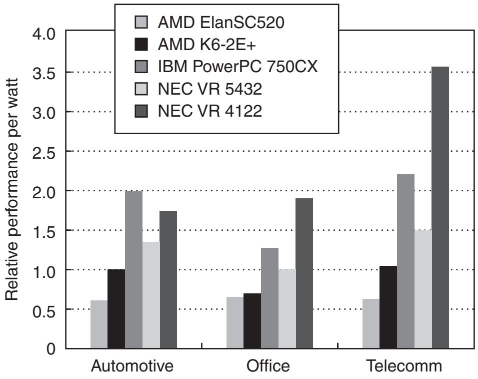


Figure E.9 Relative performance per watt for the five embedded processors. The power is measured as typical operating power for the processor and does not include any interface chips.

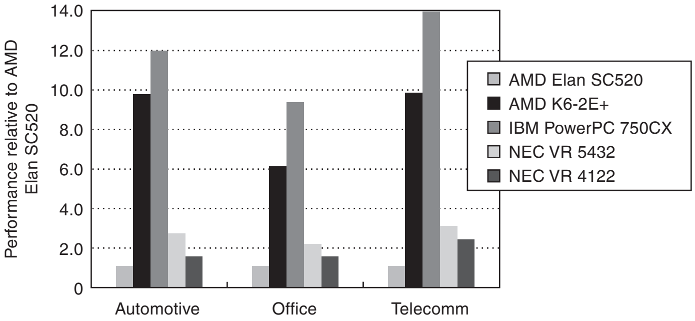


Figure E.10 Raw performance for the five embedded processors. The performance is presented as relative to the performance of the AMD ElanSC520.

E.4 Embedded Multiprocessors

Multiprocessors are now common in server environments, and several desktop multiprocessors are available from vendors, such as Sun, Compaq, and Apple. In the embedded space, a number of special-purpose designs have used customized multiprocessors, including the Sony PlayStation 2 (see Section E.5).

Many special-purpose embedded designs consist of a general-purpose programmable processor or DSP with special-purpose, finite-state machines that are used for stream-oriented I/O. In applications ranging from computer graphics and media processing to telecommunications, this style of special-purpose multiprocessor is becoming common. Although the interprocessor interactions in such designs are highly regimented and relatively simple—consisting primarily of a simple communication channel—because much of the design is committed to silicon, ensuring that the communication protocols among the input/output processors and the general-purpose processor are correct is a major challenge in such designs.

More recently, we have seen the first appearance, in the embedded space, of embedded multiprocessors built from several general-purpose processors. These multiprocessors have been focused primarily on the high-end telecommunications and networking market, where scalability is critical. An example of such a design is the MXP processor designed by empowerTel Networks for use in voice-over-IP systems. The MXP processor consists of four main components:

- An interface to serial voice streams, including support for handling jitter
- Support for fast packet routing and channel lookup
- A complete Ethernet interface, including the MAC layer
- Four MIPS32 R4000-class processors, each with its own cache (a total of 48 KB or 12 KB per processor)

The MIPS processors are used to run the code responsible for maintaining the voice-over-IP channels, including the assurance of quality of service, echo cancellation, simple compression, and packet encoding. Since the goal is to run as many independent voice streams as possible, a multiprocessor is an ideal solution.

Because of the small size of the MIPS cores, the entire chip takes only 13.5 M transistors. Future generations of the chip are expected to handle more voice channels, as well as do more sophisticated echo cancellation, voice activity detection, and more sophisticated compression.

Multiprocessing is becoming widespread in the embedded computing arena for two primary reasons. First, the issues of binary software compatibility, which plague desktop and server systems, are less relevant in the embedded space. Often software in an embedded application is written from scratch for an application or significantly modified (note that this is also the reason VLIW is favored over superscalar in embedded instruction-level parallelism). Second, the applications often have natural parallelism, especially at the high end of the embedded space. Examples of this natural parallelism abound in applications such as a settop box, a network switch, a cell phone (see Section E.7) or a game system (see Section E.5). The lower barriers to use of thread-level parallelism together with the greater sensitivity to die cost (and hence efficient use of silicon) are leading to widespread adoption of multiprocessing in the embedded space, as the application needs grow to demand more performance.

E.5 Case Study: The Emotion Engine of the Sony PlayStation 2

Desktop computers and servers rely on the memory hierarchy to reduce average access time to relatively static data, but there are embedded applications where data are often a continuous stream. In such applications there is still spatial locality, but temporal locality is much more limited.

To give another look at memory performance beyond the desktop, this section examines the microprocessor at the heart of the Sony PlayStation 2. As we will see, the steady stream of graphics and audio demanded by electronic games leads to a different approach to memory design. The style is high bandwidth via many dedicated independent memories.

Figure E.11 shows a block diagram of the Sony PlayStation 2 (PS2). Not surprisingly for a game machine, there are interfaces for video, sound, and a DVD player. Surprisingly, there are two standard computer I/O buses, USB and IEEE 1394, a PCMCIA slot as found in portable PCs, and a modem. These additions show that Sony had greater plans for the PS2 beyond traditional games. Although it appears that the I/O processor (IOP) simply handles the I/O devices and the game console, it includes a 34 MHz MIPS processor that also acts as the emulation computer to run games for earlier Sony PlayStations. It also connects to a standard PC audio card to provide the sound for the games.

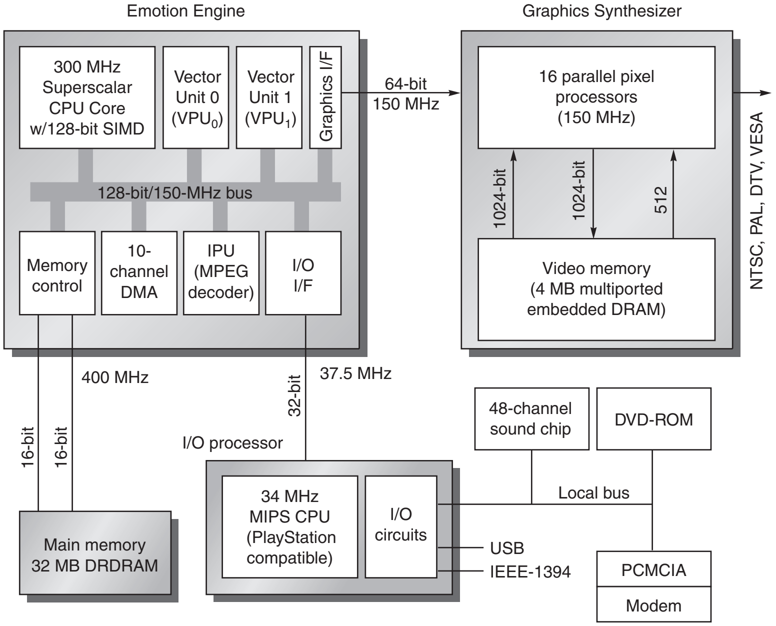

Emotion Engine
300 MHz Superscalar CPU Core w/128-bit SIMD
Vector Unit 0 (VPU$_0$)
Vector Unit 1 (VPU$_1$)
Graphics I/F
128-bit/150-MHz bus
Memory control
10-channel DMA
IPU (MPEG decoder)
I/O I/F
64-bit
150 MHz
Graphics Synthesizer
16 parallel pixel processors (150 MHz)
1024-bit
1024-bit
512
Video memory (4 MB multiported embedded DRAM)
NTSC, PAL, DTV, VESA
400 MHz
16-bit
16-bit
Main memory 32 MB DRDRAM
37.5 MHz
32-bit
I/O processor
34 MHz MIPS CPU (PlayStation compatible)
I/O circuits
48-channel sound chip
DVD-ROM
Local bus
USB
IEEE-1394
PCMCIA
Modem

Figure E.11 Block diagram of the Sony PlayStation 2. The 10 DMA channels orchestrate the transfers between all the small memories on the chip, which when completed all head toward the Graphics Interface so as to be rendered by the Graphics Synthesizer. The Graphics Synthesizer uses DRAM on chip to provide an entire frame buffer plus graphics processors to perform the rendering desired based on the display commands given from the Emotion Engine. The embedded DRAM allows 1024-bit transfers between the pixel processors and the display buffer. The Superscalar CPU is a 64-bit MIPS III with two-instruction issue, and comes with a two-way, set associative, 16 KB instruction cache; a two-way, set associative, 8 KB data cache; and 16 KB of scratchpad memory. It has been extended with 128-bit SIMD instructions for multimedia applications (see Section E.2). Vector Unit 0 is primarily a DSP-like coprocessor for the CPU (see Section E.2), which can operate on 128-bit registers in SIMD manner between 8 bits and 32 bits per word. It has 4 KB of instruction memory and 4 KB of data memory. Vector Unit 1 has similar functions to VPU0, but it normally operates independently of the CPU and contains 16 KB of instruction memory and 16 KB of data memory. All three units can communicate over the 128-bit system bus, but there is also a 128-bit dedicated path between the CPU and VPU0 and a 128-bit dedicated path between VPU1 and the Graphics Interface. Although VPU0 and VPU1 have identical microarchitectures, the differences in memory size and units to which they have direct connections affect the roles that they take in a game. At 0.25-micron line widths, the Emotion Engine chip uses 13.5M transistors and is 225 mm^2, and the Graphics Synthesizer is 279 mm^2. To put this in perspective, the Alpha 21264 microprocessor in 0.25-micron technology is about 160 mm^2 and uses 15M transistors. (This figure is based on Figure 1 in "Sony's Emotionally Charged Chip," *Microprocessor Report* 13:5.)

Thus, one challenge for the memory system of this embedded application is to act as source or destination for the extensive number of I/O devices. The PS2 designers met this challenge with two PC800 (400 MHz) DRDRAM chips using two channels, offering 32 MB of storage and a peak memory bandwidth of 3.2 GB/sec.

What's left in the figure are basically two big chips: the Graphics Synthesizer and the Emotion Engine.

The Graphics Synthesizer takes rendering commands from the Emotion Engine in what are commonly called *display lists.* These are lists of 32-bit commands that tell the renderer what shape to use and where to place them, plus what colors and textures to fill them.

This chip also has the highest bandwidth portion of the memory system. By using embedded DRAM on the Graphics Synthesizer, the chip contains the full video buffer *and* has a 2048-bit-wide interface so that pixel filling is not a bottleneck. This embedded DRAM greatly reduces the bandwidth demands on the DRDRAM. It illustrates a common technique found in embedded applications: separate memories dedicated to individual functions to inexpensively achieve greater memory bandwidth for the entire system.

The remaining large chip is the Emotion Engine, and its job is to accept inputs from the IOP and create the display lists of a video game to enable 3D video transformations in real time. A major insight shaped the design of the Emotion Engine: Generally, in a racing car game there are foreground objects that are constantly changing and background objects that change less in reaction to the events, although the background can be most of the screen. This observation led to a split of responsibilities.

The CPU works with VPU0 as a tightly coupled coprocessor, in that every VPU0 instruction is a standard MIPS coprocessor instruction, and the addresses are generated by the MIPS CPU. VPU0 is called a vector processor, but it is similar to 128-bit SIMD extensions for multimedia found in several desktop processors (see Section E.2).

VPU1, in contrast, fetches its own instructions and data and acts in parallel with CPU/VPU0, acting more like a traditional vector unit. With this split, the more flexible CPU/VPU0 handles the foreground action and the VPU1 handles the background. Both deposit their resulting display lists into the Graphics Interface to send the lists to the Graphics Synthesizer.

Thus, the programmers of the Emotion Engine have three processor sets to choose from to implement their programs: the traditional 64-bit MIPS architecture including a floating-point unit, the MIPS architecture extended with multimedia instructions (VPU0), and an independent vector processor (VPU1). To accelerate MPEG decoding, there is another coprocessor (Image Processing Unit) that can act independent of the other two.

With this split of function, the question then is how to connect the units together, how to make the data flow between units, and how to provide the memory bandwidth needed by all these units. As mentioned earlier, the Emotion Engine designers chose many dedicated memories. The CPU has a 16 KB scratch pad memory (SPRAM) in addition to a 16 KB instruction cache and an 8 KB data cache. VPU0 has a 4 KB instruction memory and a 4 KB data memory, and VPU1 has a 16 KB instruction memory and a 16 KB data memory. Note that these are four *memories*, not caches of a larger memory elsewhere. In each memory the latency is just 1 clock cycle. VPU1 has more memory than VPU0 because it creates the bulk of the display lists and because it largely acts independently.

The programmer organizes all memories as two double buffers, one pair for the incoming DMA data and one pair for the outgoing DMA data. The programmer then uses the various processors to transform the data from the input buffer to the output buffer. To keep the data flowing among the units, the programmer next sets up the 10 DMA channels, taking care to meet the real-time deadline for realistic animation of 15 frames per second.

Figure E.12 shows that this organization supports two main operating modes: serial, where CPU/VPU0 acts as a preprocessor on what to give VPU1 for it to create for the Graphics Interface using the scratchpad memory as the buffer, and parallel, where both the CPU/VPU0 and VPU1 create display lists. The display lists and the Graphics Synthesizer have multiple context identifiers to distinguish the parallel display lists to produce a coherent final image.

All units in the Emotion Engine are linked by a common 150 MHz, 128-bit-wide bus. To offer greater bandwidth, there are also two dedicated buses: a 128-bit path between the CPU and VPU0 and a 128-bit path between VPU1 and the Graphics Interface. The programmer also chooses which bus to use when setting up the DMA channels.

Looking at the big picture, if a server-oriented designer had been given the problem, we might see a single common bus with many local caches and cache-coherent mechanisms to keep data consistent. In contrast, the PlayStation 2 followed the tradition of embedded designers and has at least nine distinct memory modules. To keep the data flowing in real time from memory to the display, the PS2 uses dedicated memories, dedicated buses, and DMA channels. Coherency is the responsibility of the programmer, and, given the continuous flow from main memory to the graphics interface and the real-time requirements, programmer-controlled coherency works well for this application.

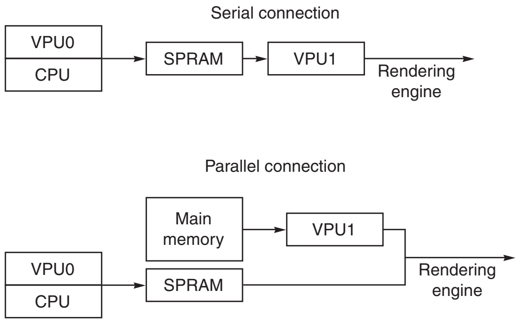


Figure E.12 Two modes of using Emotion Engine organization. The first mode divides the work between the two units and then allows the Graphics Interface to properly merge the display lists. The second mode uses CPU/VPU0 as a filter of what to send to VPU1, which then does all the display lists. It is up to the programmer to choose between serial and parallel data flow. SPRAM is the scratchpad memory.

E.6 Case Study: Sanyo VPC-SX500 Digital Camera

Another very familiar embedded system is a digital camera. Here we consider the Sanyo VPC-SX500. When powered on, the microprocessor of the camera first runs diagnostics on all components and writes any error messages to the liquid crystal display (LCD) on the back of the camera. This camera uses a 1.8-inch low-temperature polysilicon thin-film transistor (TFT) color LCD. When a photographer takes a picture, he first holds the shutter halfway so that the microprocessor can take a light reading. The microprocessor then keeps the shutter open to get the necessary light, which is captured by a charge-coupled device (CCD) as red, green, and blue pixels. The CCD is a 1/2-inch, 1360 × 1024-pixel, progressive-scan chip. The pixels are scanned out row by row; passed through routines for white balance, color, and aliasing correction; and then stored in a 4 MB frame buffer. The next step is to compress the image into a standard format, such as JPEG, and store it in the removable Flash memory. The photographer picks the compression, in this camera called either *fine* or *normal*, with a compression ratio of 10 to 20 times. A 512 MB Flash memory can store at least 1200 fine-quality compressed images or approximately 2000 normal-quality compressed images. The microprocessor then updates the LCD display to show that there is room for one less picture.

Although the previous paragraph covers the basics of a digital camera, there are many more features that are included: showing the recorded images on the color LCD display, sleep mode to save battery life, monitoring battery energy, buffering to allow recording a rapid sequence of uncompressed images, and, in this camera, video recording using MPEG format and audio recording using WAV format.

The electronic brain of this camera is an embedded computer with several special functions embedded on the chip [Okada et al. 1999]. Figure E.13 shows the block diagram of a chip similar to the one in the camera. As mentioned in Section E.1, such chips have been called *systems on a chip* (SOCs) because they essentially integrate into a single chip all the parts that were found on a small printed circuit board of the past. A SOC generally reduces size and lowers power compared to less integrated solutions. Sanyo claims their SOC enables the camera to operate on half the number of batteries and to offer a smaller form factor than competitors' cameras. For higher performance, it has two buses. The 16-bit bus is for the many slower I/O devices: SmartMedia interface, program and data memory, and DMA. The 32-bit bus is for the SDRAM, the signal processor (which is connected to the CCD), the Motion JPEG encoder, and the NTSC/PAL encoder (which is connected to the LCD). Unlike desktop microprocessors, note the large variety of I/O buses that this chip must integrate. The 32-bit RISC MPU is a proprietary design and runs at 28.8 MHz, the same clock rate as the buses. This 700 mW chip contains 1.8M transistors in a 10.5 × 10.5 mm die implemented using a 0.35-micron process.

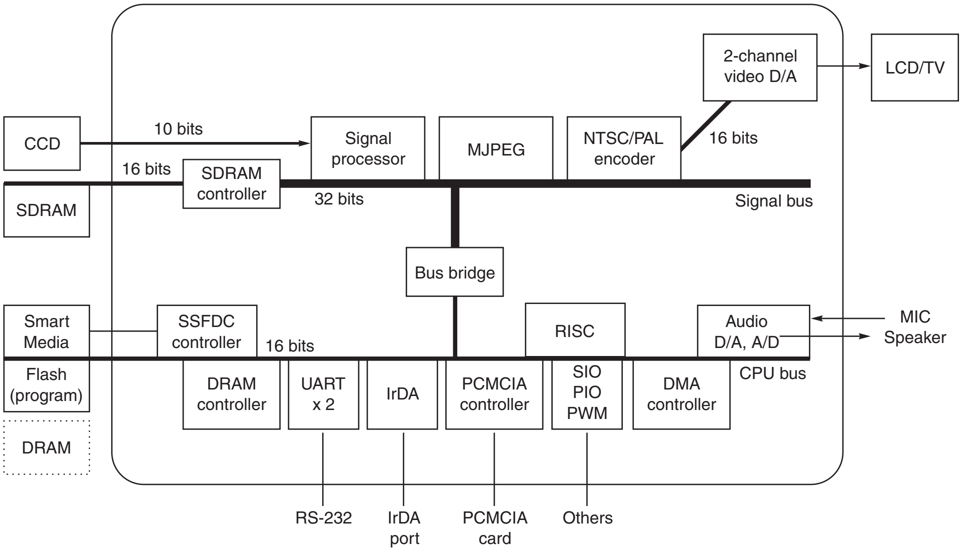

2-channel video D/A
LCD/TV
CCD
10 bits
Signal processor
MJPEG
NTSC/PAL encoder
16 bits
16 bits
SDRAM controller
SDRAM
32 bits
Signal bus
Bus bridge
Smart Media
SSFDC controller
16 bits
RISC
Audio D/A, A/D
MIC
Speaker
Flash (program)
DRAM controller
UART x 2
IrDA
PCMCIA controller
SIO PIO PWM
DMA controller
CPU bus
DRAM
RS-232
IrDA port
PCMCIA card
Others

Figure E.13 The system on a chip (SOC) found in Sanyo digital cameras. This block diagram, found in Okada et al. [1999], is for the predecessor of the SOC in the camera described in the text. The successor SOC, called *Super Advanced IC*, uses three buses instead of two, operates at 60 MHz, consumes 800 mW, and fits 3.1M transistors in a 10.2 × 10.2 mm die using a 0.35-micron process. Note that this embedded system has twice as many transistors as the state-of-the-art, high-performance microprocessor in 1990! The SOC in the figure is limited to processing 1024 × 768 pixels, but its successor supports 1360 × 1024 pixels.

E.7 Case Study: Inside a Cell Phone

Although gaming consoles and digital cameras are familiar embedded systems, today the most familiar embedded system is the cell phone. In 1999, there were 76 million cellular subscribers in the United States, a 25% growth rate from the year before. That growth rate is almost 35% per year worldwide, as developing countries find it much cheaper to install cellular towers than copper-wire-based infrastructure. Thus, in many countries, the number of cell phones in use exceeds the number of wired phones in use.

Not surprisingly, the cellular handset market is growing at 35% per year, with about 280 million cellular phone handsets sold worldwide in 1999. To put that in perspective, in the same year sales of personal computers were 120 million. These numbers mean that tremendous engineering resources are available to improve cell phones, and cell phones are probably leaders in engineering innovation per cubic inch [Grice and Kanellos 2000].

Before unveiling the anatomy of a cell phone, let’s try a short introduction to wireless technology.

Background on Wireless Networks

Networks can be created out of thin air as well as out of copper and glass, creating *wireless networks.* Much of this section is based on a report from the National Research Council [1997].

A radio wave is an electromagnetic wave propagated by an antenna. Radio waves are modulated, which means that the sound signal is superimposed on the stronger radio wave that carries the sound signal, and hence is called the *carrier signal.* Radio waves have a particular wavelength or frequency: They are measured either as the length of the complete wave or as the number of waves per second. Long waves have low frequencies, and short waves have high frequencies. FM radio stations transmit on the band of 88 MHz to 108 MHz using frequency modulations (FM) to record the sound signal.

By tuning in to different frequencies, a radio receiver can pick up a specific signal. In addition to AM and FM radio, other frequencies are reserved for citizens band radio, television, pagers, air traffic control radar, Global Positioning System, and so on. In the United States, the Federal Communications Commission decides who gets to use which frequencies and for what purpose.

The *bit error rate* (BER) of a wireless link is determined by the received signal power, noise due to interference caused by the receiver hardware, interference from other sources, and characteristics of the channel. Noise is typically proportional to the radio frequency bandwidth, and a key measure is the *signal-to-noise ratio* (SNR) required to achieve a given BER. Figure E.14 lists more challenges for wireless communication.

Typically, wireless communication is selected because the communicating devices are mobile or because wiring is inconvenient, which means the wireless network must rearrange itself dynamically. Such rearrangement makes routing

Challenge	Description	Impact
Path loss	Received power divided by transmitted power; the radio must overcome signal-to-noise ratio (SNR) of noise from interference. Path loss is exponential in distance and depends on interference if it is above 100 meters.	1 W transmit power, 1 GHz transmit frequency, 1 Mbit/sec data rate at 10^{-7} BER, distance between radios can be 728 meters in free space vs. 4 meters in a dense jungle.
Shadow fading	Received signal blocked by objects, buildings outdoors, or walls indoors; increase power to improve received SNR. It depends on the number of objects and their dielectric properties.	If transmitter is moving, need to change transmit power to ensure received SNR in region.
Multipath fading	Interference between multiple versions of signal that arrive at different times, determined by time between fastest signal and slowest signal relative to signal bandwidth.	900 MHz transmit frequency signal power changes every 30 cm.
Interference	Frequency reuse, adjacent channel, narrow band interference.	Requires filters, spread spectrum.

Figure E.14 Challenges for wireless communication.

more challenging. A second challenge is that wireless signals are not protected and hence are subject to mutual interference, especially as devices move. Power is another challenge for wireless communication, both because the devices tend to be battery powered and because antennas radiate power to communicate and little of it reaches the receiver. As a result, raw bit error rates are typically a thousand to a million times higher than copper wire.

There are two primary architectures for wireless networks: *base station* architectures and *peer-to-peer* architectures. Base stations are connected by landlines for longer-distance communication, and the mobile units communicate only with a single local base station. Peer-to-peer architectures allow mobile units to communicate with each other, and messages hop from one unit to the next until delivered to the desired unit. Although peer-to-peer is more reconfigurable, base stations tend to be more reliable since there is only one hop between the device and the station. *Cellular telephony*, the most popular example of wireless networks, relies on radio with base stations.

Cellular systems exploit exponential path loss to reuse the same frequency at spatially separated locations, thereby greatly increasing the number of customers served. Cellular systems will divide a city into nonoverlapping hexagonal cells that use different frequencies if nearby, reusing a frequency only when cells are far enough apart so that mutual interference is acceptable.

At the intersection of three hexagonal cells is a base station with transmitters and antennas that is connected to a switching office that coordinates handoffs when a mobile device leaves one cell and goes into another, as well as accepts and places calls over landlines. Depending on topography, population, and so on, the radius of a typical cell is 2 to 10 miles.

The Cell Phone

Figure E.15 shows the components of a radio, which is the heart of a cell phone. Radio signals are first received by the antenna, amplified, passed through a mixer, then filtered, demodulated, and finally decoded. The antenna acts as the interface between the medium through which radio waves travel and the electronics of the transmitter or receiver. Antennas can be designed to work best in particular directions, giving both transmission and reception directional properties. Modulation encodes information in the amplitude, phase, or frequency of the signal to increase its robustness under impaired conditions. Radio transmitters go through the same steps, just in the opposite order.

Originally, all components were analog, but over time most were replaced by digital components, requiring the radio signal to be converted from analog to digital. The desire for flexibility in the number of radio bands led to software routines replacing some of these functions in programmable chips, such as digital signal processors. Because such processors are typically found in mobile devices, emphasis is placed on performance per joule to extend battery life, performance per square millimeter of silicon to reduce size and cost, and bytes per task to reduce memory size.

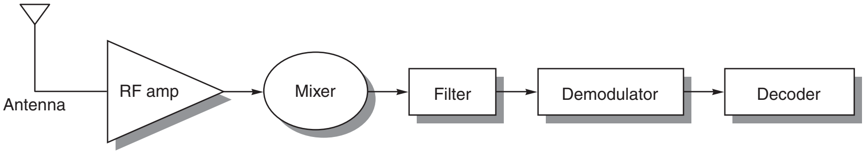


Figure E.15 A radio receiver consists of an antenna, radio frequency amplifier, mixer, filters, demodulator, and decoder. A mixer accepts two signal inputs and forms an output signal at the sum and difference frequencies. Filters select a narrower band of frequencies to pass on to the next stage. Modulation encodes information to make it more robust. Decoding turns signals into information. Depending on the application, all electrical components can be either analog or digital. For example, a car radio is all analog components, but a PC modem is all digital except for the amplifier. Today analog silicon chips are used for the RF amplifier and first mixer in cellular phones.

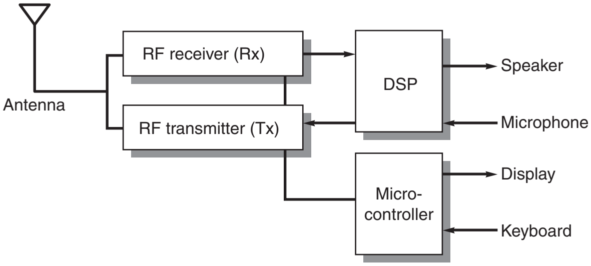


Figure E.16 Block diagram of a cell phone. The DSP performs the signal processing steps of Figure E.15, and the microcontroller controls the user interface, battery management, and call setup. (Based on Figure 1.3 of Groe and Larson [2000].)

Figure E.16 shows the generic block diagram of the electronics of a cell phone handset, with the DSP performing the signal processing and the microcontroller handling the rest of the tasks. Cell phone handsets are basically mobile computers acting as a radio. They include standard I/O devices—keyboard and LCD display—plus a microphone, speaker, and antenna for wireless networking. Battery efficiency affects sales, both for standby power when waiting for a call and for minutes of speaking.

When a cell phone is turned on, the first task is to find a cell. It scans the full bandwidth to find the strongest signal, which it keeps doing every seven seconds or if the signal strength drops, since it is designed to work from moving vehicles. It then picks an unused radio channel. The local switching office registers the cell phone and records its phone number and electronic serial number, and assigns it a voice channel for the phone conversation. To be sure the cell phone got the right channel, the base station sends a special tone on it, which the cell phone sends back to acknowledge it. The cell phone times out after 5 seconds if it doesn't hear the supervisory tone, and it starts the process all over again. The original base station makes a handoff request to the incoming base station as the signal strength drops offs.

To achieve a two-way conversation over radio, frequency bands are set aside for each direction, forming a frequency pair or *channel.* The original cellular base stations transmitted at 869.04 to 893.97 MHz (called the *forward path*), and cell phones transmitted at 824.04 to 848.97 MHz (called t*he reverse path*), with the frequency gap to keep them from interfering with each other. Cells might have had between 4 and 80 channels. Channels were divided into setup channels for call setup and voice channels to handle the data or voice traffic.

The communication is done digitally, just like a modem, at 9600 bits/sec. Since wireless is a lossy medium, especially from a moving vehicle, the handset sends each message five times. To preserve battery life, the original cell phones typically transmit at two signal strengths—0.6 W and 3.0 W—depending on the distance to the cell. This relatively low power not only allows smaller batteries and thus smaller cell phones, but it also aids frequency reuse, which is the key to cellular telephony.

Figure E.17 shows a circuit board from a Nokia digital phone, with the components identified. Note that the board contains two processors. A Z-80 microcontroller is responsible for controlling the functions of the board, I/O with the keyboard and display, and coordinating with the base station. The DSP handles all signal compression and decompression. In addition there are dedicated chips for analog-to-digital and digital-to-analog conversion, amplifiers, power management, and RF interfaces.

In 2001, a cell phone had about 10 integrated circuits, including parts made in exotic technologies like gallium arsinide and silicon germanium as well as standard CMOS. The economics and desire for flexibility have shrunk this to just a few chips. However, these SOCs still contain a separate microcontroller and DSP, with code implementing many of the functions just described.

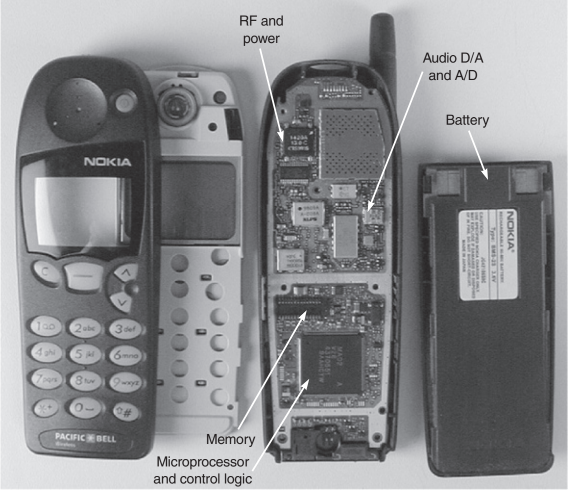


Figure E.17 Circuit board from a Nokia cell phone. (Courtesy HowStuffWorks, Inc.)

Cell Phone Standards and Evolution

Improved communication speeds for cell phones were developed with multiple standards. *Code division multiple access* (CDMA), as one popular example, uses a wider radio frequency band for a path than the original cell phones, called *advanced mobile phone service* (AMPS), a mostly analog system. The wider frequency makes it more difficult to block and is called *spread spectrum.* Other standards are *time division multiple access* (TDMA) and *global system for mobile communication* (GSM). These second-generation standards—CDMA, GSM, and TDMA—are mostly digital.

The big difference for CDMA is that all callers share the same channel, which operates at a much higher rate, and it then distinguishes the different calls by encoding each one uniquely. Each CDMA phone call starts at 9600 bits/sec; it is then encoded and transmitted as equal-sized messages at 1.25 Mbits/sec. Rather than send each signal five times as in AMPS, each bit is stretched so that it takes 11 times the minimum frequency, thereby accommodating interference and yet successful transmission. The base station receives the messages, and it separates them into the separate 9600 bit/sec streams for each call.

To enhance privacy, CDMA uses pseudorandom sequences from a set of 64 predefined codes. To synchronize the handset and base station so as to pick a common pseudorandom seed, CDMA relies on a clock from the Global Positioning System, which continuously transmits an accurate time signal. By carefully selecting the codes, the shared traffic sounds like random noise to the listener. Hence, as more users share a channel there is more noise, and the signal-to-noise ratio gradually degrades. Thus, the capacity of the CDMA system is a matter of taste, depending upon the sensitivity of the listener to background noise.

In addition, CDMA uses speech compression and varies the rate of data transferred depending upon how much activity is going on in the call. Both these techniques preserve bandwidth, which allows for more calls per cell. CDMA must regulate power carefully so that signals near the cell tower do not overwhelm those from far away, with the goal of all signals reaching the tower at about the same level. The side benefit is that CDMA handsets emit less power, which both helps battery life and increases capacity when users are close to the tower.

Thus, compared to AMPS, CDMA improves the capacity of a system by up to an order of magnitude, has better call quality, has better battery life, and enhances users' privacy. After considerable commercial turmoil, there is a new third-generation standard called *International Mobile Telephony 2000* (IMT-2000), based primarily on two competing versions of CDMA and one TDMA. This standard may lead to cell phones that work anywhere in the world.

E.8 Concluding Remarks

Embedded systems are a very broad category of computing devices. This appendix has shown just some aspects of this. For example, the TI 320C55 DSP is a relatively "RISC-like" processor designed for embedded applications, with very fine-tuned capabilities. On the other end of the spectrum, the TI 320C64x is a very high-performance, eight-issue VLIW processor for very demanding tasks. Some processors must operate on battery power alone; others have the luxury of being plugged into line current. Unifying all of these is a need to perform some level of signal processing for embedded applications. Media extensions attempt to merge DSPs with some more general-purpose processing abilities to make these processors usable for signal processing applications. We examined several case studies, including the Sony PlayStation 2, digital cameras, and cell phones. The PS2 performs detailed three-dimensional graphics, whereas a cell phone encodes and decodes signals according to elaborate communication standards. But both have system architectures that are very different from general-purpose desktop or server platforms. In general, architectural decisions that seem practical for general-purpose applications, such as multiple levels of caching or out-of-order superscalar execution, are much less desirable in embedded applications. This is due to chip area, cost, power, and real-time constraints. The programming model that these systems present places more demands on both the programmer and the compiler for extracting parallelism.

F.1 Introduction F-2
F.2 Interconnecting Two Devices F-6
F.3 Connecting More Than Two Devices F-20
F.4 Network Topology F-30
F.5 Network Routing, Arbitration, and Switching F-44
F.6 Switch Microarchitecture F-56
F.7 Practical Issues for Commercial Interconnection Networks F-66
F.8 Examples of Interconnection Networks F-73
F.9 Internetworking F-85
F.10 Crosscutting Issues for Interconnection Networks F-89
F.11 Fallacies and Pitfalls F-92
F.12 Concluding Remarks F-100
F.13 Historical Perspective and References F-101
References F-109
Exercises F-111

F

Interconnection Networks

Revised by Timothy M. Pinkston, University of Southern California;
José Duato, Universitat Politècnica de València, and Simula

"The Medium is the Message" because it is the medium that shapes and controls the search and form of human associations and actions.

Marshall McLuhan
Understanding Media (1964)

The marvels—of film, radio, and television—are marvels of one-way communication, which is not communication at all.

Milton Mayer
On the Remote Possibility of
Communication (1967)

The interconnection network is the heart of parallel architecture.

Chuan-Lin Wu and Tse-Yun Feng
Interconnection Networks for Parallel
and Distributed Processing (1984)

Indeed, as system complexity and integration continues to increase, many designers are finding it more efficient to route packets, not wires.

Bill Dally
Principles and Practices of
Interconnection Networks (2004)

F.1 Introduction

Previous chapters and appendices cover the components of a single computer but give little consideration to the interconnection of those components and how multiple computer systems are interconnected. These aspects of computer architecture have gained significant importance in recent years. In this appendix we see how to connect individual devices together into a community of communicating devices, where the term *device* is generically used to signify anything from a component or set of components within a computer to a single computer to a system of computers. Figure F.1 shows the various elements comprising this community: end nodes consisting of devices and their associated hardware and software interfaces, links from end nodes to the interconnection network, and the interconnection network. Interconnection networks are also called *networks, communication subnets*, or *communication subsystems*. The interconnection of multiple networks is called *internetworking*. This relies on communication standards to convert information from one kind of network to another, such as with the Internet.

There are several reasons why computer architects should devote attention to interconnection networks. In addition to providing external connectivity, networks are commonly used to interconnect the components within a single computer at many levels, including the processor microarchitecture. Networks have long been used in mainframes, but today such designs can be found in personal computers as well, given the high demand on communication bandwidth needed to enable increased computing power and storage capacity. Switched networks are replacing buses as the normal means of communication between computers, between I/O devices, between boards, between chips, and even between modules inside chips. Computer architects must understand interconnect problems and solutions in order to more effectively design and evaluate computer systems.

Interconnection networks cover a wide range of application domains, very much like memory hierarchy covers a wide range of speeds and sizes. Networks implemented within processor chips and systems tend to share characteristics much in common with processors and memory, relying more on high-speed hardware solutions and less on a flexible software stack. Networks implemented across systems tend to share much in common with storage and I/O, relying more on the operating system and software protocols than high-speed hardware—though we are seeing a convergence these days. Across the domains, performance includes latency and effective bandwidth, and queuing theory is a valuable analytical tool in evaluating performance, along with simulation techniques.

This topic is vast—portions of Figure F.1 are the subject of entire books and college courses. The goal of this appendix is to provide for the computer architect an overview of network problems and solutions. This appendix gives introductory explanations of key concepts and ideas, presents architectural implications of interconnection network technology and techniques, and provides useful references to more detailed descriptions. It also gives a common framework for evaluating all types of interconnection networks, using a single set of terms to describe the basic

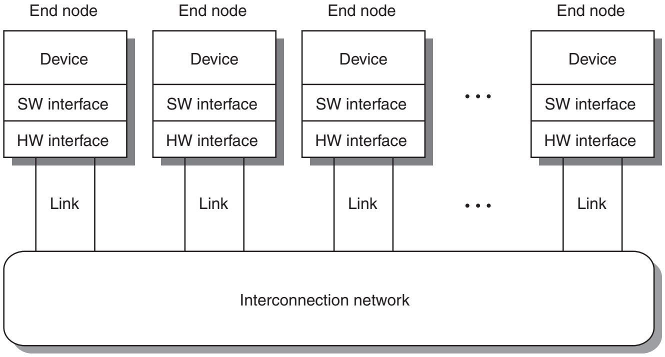

End node
End node
End node
End node
Device
Device
Device
Device
SW interface
SW interface
SW interface
SW interface
HW interface
HW interface
HW interface
HW interface
Link
Link
Link
Link
Interconnection network

Figure F.1 A conceptual illustration of an interconnected community of devices.

alternatives. As we will see, many types of networks have common preferred alternatives, but for others the best solutions are quite different. These differences become very apparent when crossing between the networking domains.

Interconnection Network Domains

Interconnection networks are designed for use at different levels within and across computer systems to meet the operational demands of various application areas—high-performance computing, storage I/O, cluster/workgroup/enterprise systems, internetworking, and so on. Depending on the number of devices to be connected and their proximity, we can group interconnection networks into four major networking domains:

- *On-chip networks* (OCNs)—Also referred to as network-on-chip (NoC), this type of network is used for interconnecting microarchitecture functional units, register files, caches, compute tiles, and processor and IP cores within chips or multichip modules. Current and near future OCNs support the connection of a few tens to a few hundred of such devices with a maximum interconnection distance on the order of centimeters. Most OCNs used in high-performance chips are custom designed to mitigate chip-crossing wire delay problems caused by increased technology scaling and transistor integration, though some proprietary designs are gaining wider use (e.g., IBM's CoreConnect, ARM's AMBA, and Sonic's Smart Interconnect). Examples of current OCNs are those found in the Intel Teraflops processor chip [Hoskote07], connecting 80 simple cores; the Intel Single-Chip Cloud Computer (SCCC) [Howard10], connecting 48 IA-32 architecture cores; and Tilera's TILE-Gx line of processors [TILE-GX], connecting 100 processing cores in 4Q 2011 using TSMC's 40 nanometer process and 200 cores planned for 2013 (code named "Stratton") using

TSMC's 28 nanometer process. The networks peak at 256 GBps for both Intel prototypes and up to 200 Tbps for the TILE-Gx100 processor. More detailed information for OCNs is provided in Flich [2010].

- *System/storage area networks* (SANs)—This type of network is used for interprocessor and processor-memory interconnections within multiprocessor and multicomputer systems, and also for the connection of storage and I/O components within server and data center environments. Typically, several hundreds of such devices can be connected, although some supercomputer SANs support the interconnection of many thousands of devices, like the IBM Blue Gene/L supercomputer. The maximum interconnection distance covers a relatively small area—on the order of a few tens of meters usually—but some SANs have distances spanning a few hundred meters. For example, *InfiniBand*, a popular SAN standard introduced in late 2000, supports system and storage I/O interconnects at up to 120 Gbps over a distance of 300 m.
- *Local area networks* (LANs)—This type of network is used for interconnecting autonomous computer systems distributed across a machine room or throughout a building or campus environment. Interconnecting PCs in a cluster is a prime example. Originally, LANs connected only up to a hundred devices, but with bridging LANs can now connect up to a few thousand devices. The maximum interconnect distance covers an area of a few kilometers usually, but some have distance spans of a few tens of kilometers. For instance, the most popular and enduring LAN, *Ethernet*, has a 10 Gbps standard version that supports maximum performance over a distance of 40 km.
- *Wide area networks* (WANs)—Also called *long-haul networks*, WANs connect computer systems distributed across the globe, which requires internetworking support. WANs connect many millions of computers over distance scales of many thousands of kilometers. Asynchronous Transfer Mode (ATM) is an example of a WAN.

Figure F.2 roughly shows the relationship of these networking domains in terms of the number of devices interconnected and their distance scales. Overlap exists for some of these networks in one or both dimensions, which leads to product competition. Some network solutions have become commercial standards while others remain proprietary. Although the preferred solutions may significantly differ from one interconnection network domain to another depending on the design requirements, the problems and concepts used to address network problems remain remarkably similar across the domains. No matter the target domain, networks should be designed so as not to be the bottleneck to system performance and cost efficiency. Hence, the ultimate goal of computer architects is to design interconnection networks of the lowest possible cost that are capable of transferring the maximum amount of available information in the shortest possible time.

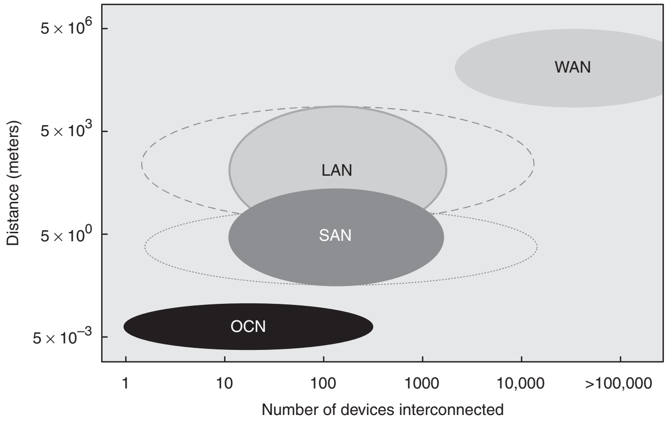


Figure F.2 Relationship of the four interconnection network domains in terms of number of devices connected and their distance scales: on-chip network (OCN), system/storage area network (SAN), local area network (LAN), and wide area network (WAN). Note that there are overlapping ranges where some of these networks compete. Some supercomputer systems use proprietary custom networks to interconnect several thousands of computers, while other systems, such as multicomputer clusters, use standard commercial networks.

Approach and Organization of This Appendix

Interconnection networks can be well understood by taking a top-down approach to unveiling the concepts and complexities involved in designing them. We do this by viewing the network initially as an opaque "black box" that simply and ideally performs certain necessary functions. Then we systematically open various layers of the black box, allowing more complex concepts and nonideal network behavior to be revealed. We begin this discussion by first considering the interconnection of just two devices in Section F.2, where the black box network can be viewed as a simple *dedicated link* network—that is, wires or collections of wires running bidirectionally between the devices. We then consider the interconnection of more than two devices in Section F.3, where the black box network can be viewed as a *shared link* network or as a *switched point-to-point* network connecting the devices. We continue to peel away various other layers of the black box by considering in more detail the network topology (Section F.4); routing, arbitration, and switching (Section F.5); and switch microarchitecture (Section F.6). Practical issues for commercial networks are considered in Section F.7, followed by examples illustrating the trade-offs for each type of network in Section F.8. Internetworking is briefly discussed in Section F.9, and additional crosscutting issues for interconnection networks are presented in Section F.10. Section F.11 gives some common fallacies and pitfalls related to interconnection networks, and Section F.12 presents some concluding remarks. Finally, we provide a brief historical perspective and some suggested reading in Section F.13.

F.2 Interconnecting Two Devices

This section introduces the basic concepts required to understand how communication between just two networked devices takes place. This includes concepts that deal with situations in which the receiver may not be ready to process incoming data from the sender and situations in which transport errors may occur. To ease understanding, the black box network at this point can be conceptualized as an ideal network that behaves as simple dedicated links between the two devices. Figure F.3 illustrates this, where unidirectional wires run from device A to device B and *vice versa*, and each end node contains a buffer to hold the data. Regardless of the network complexity, whether dedicated link or not, a connection exists from each end node device to the network to inject and receive information to/from the network. We first describe the basic functions that must be performed at the end nodes to commence and complete communication, and then we discuss network media and the basic functions that must be performed by the network to carry out communication. Later, a simple performance model is given, along with several examples to highlight implications of key network parameters.

Network Interface Functions: Composing and Processing Messages

Suppose we want two networked devices to read a word from each other's memory. The unit of information sent or received is called a *message*. To acquire the desired data, the two devices must first compose and send a certain type of message in the form of a *request* containing the address of the data within the other device. The address (i.e., memory or operand location) allows the receiver to identify where to find the information being requested. After processing the request, each device then composes and sends another type of message, a *reply*, containing the data. The address and data information is typically referred to as the message *payload.*

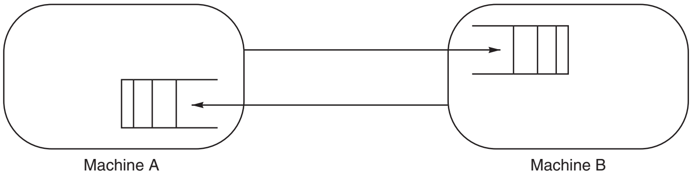


Figure F.3 A simple dedicated link network bidirectionally interconnecting two devices.

In addition to payload, every message contains some control bits needed by the network to deliver the message and process it at the receiver. The most typical are bits to distinguish between different types of messages (e.g., request, reply, request acknowledge, reply acknowledge) and bits that allow the network to transport the information properly to the destination. These additional control bits are encoded in the *header* and/or *trailer* portions of the message, depending on their location relative to the message payload. As an example, Figure F.4 shows the format of a message for the simple dedicated link network shown in Figure F.3. This example shows a single-word payload, but messages in some interconnection networks can include several thousands of words.

Before message transport over the network occurs, messages have to be composed. Likewise, upon receipt from the network, they must be processed. These and other functions described below are the role of the *network interface* (also referred to as the *channel adapter*) residing at the end nodes. Together with some direct memory access (DMA) engine and link drivers to transmit/receive messages to/from the network, some dedicated memory or register(s) may be used to buffer outgoing and incoming messages. Depending on the network domain and design specifications for the network, the network interface hardware may consist of nothing more than the communicating device itself (i.e., for OCNs and some SANs) or a separate card that integrates several embedded processors and DMA engines with thousands of megabytes of RAM (i.e., for many SANs and most LANs and WANs).

In addition to hardware, network interfaces can include software or firmware to perform the needed operations. Even the simple example shown in Figure F.3 may invoke messaging software to translate requests and replies into messages with the appropriate headers. This way, user applications need not worry about composing and processing messages as these tasks can be performed automatically at a lower level. An application program usually cooperates with the operating or runtime

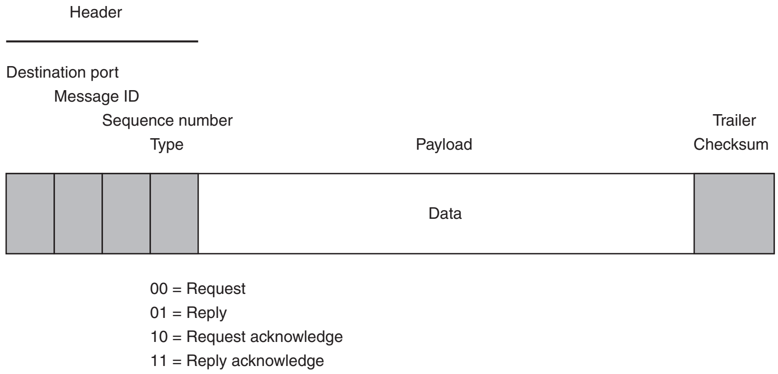


Figure F.4 An example packet format with header, payload, and checksum in the trailer.

system to send and receive messages. As the network is likely to be shared by many processes running on each device, the operating system cannot allow messages intended for one process to be received by another. Thus, the messaging software must include protection mechanisms that distinguish between processes. This distinction could be made by expanding the header with a *port* number that is known by both the sender and intended receiver processes.

In addition to composing and processing messages, additional functions need to be performed by the end nodes to establish communication among the communicating devices. Although hardware support can reduce the amount of work, some can be done by software. For example, most networks specify a maximum amount of information that can be transferred (i.e., *maximum transfer unit*) so that network buffers can be dimensioned appropriately. Messages longer than the maximum transfer unit are divided into smaller units, called *packets* (or *datagrams*), that are transported over the network. Packets are reassembled into messages at the destination end node before delivery to the application. Packets belonging to the same message can be distinguished from others by including a *message ID* field in the packet header. If packets arrive out of order at the destination, they are reordered when reassembled into a message. Another field in the packet header containing a *sequence number* is usually used for this purpose.

The sequence of steps the end node follows to commence and complete communication over the network is called a *communication protocol*. It generally has symmetric but reversed steps between sending and receiving information. Communication protocols are implemented by a combination of software and hardware to accelerate execution. For instance, many network interface cards implement hardware timers as well as hardware support to split messages into packets and reassemble them, compute the cyclic redundancy check (CRC) *checksum,* handle virtual memory addresses, and so on.

Some network interfaces include extra hardware to offload protocol processing from the host computer, such as TCP *offload engines* for LANs and WANs. But, for interconnection networks such as SANs that have low latency requirements, this may not be enough even when lighter-weight communication protocols are used such as message passing interface (MPI). Communication performance can be further improved by bypassing the operating system (OS). OS bypassing can be implemented by directly allocating message buffers in the network interface memory so that applications directly write into and read from those buffers. This avoids extra memory-to-memory copies. The corresponding protocols are referred to as *zero-copy* protocols or *user-level communication* protocols. Protection can still be maintained by calling the OS to allocate those buffers at initialization and preventing unauthorized memory accesses in hardware.

In general, some or all of the following are the steps needed to send a message at end node devices over a network:

1. The application executes a system call, which copies data to be sent into an operating system or network interface buffer, divides the message into packets (if needed), and composes the header and trailer for packets.

2. The checksum is calculated and included in the header or trailer of packets.
3. The timer is started, and the network interface hardware sends the packets.

Message reception is in the reverse order:

3. The network interface hardware receives the packets and puts them into its buffer or the operating system buffer.
2. The checksum is calculated for each packet. If the checksum matches the sender's checksum, the receiver sends an acknowledgment back to the packet sender. If not, it deletes the packet, assuming that the sender will resend the packet when the associated timer expires.
1. Once all packets pass the test, the system reassembles the message, copies the data to the user's address space, and signals the corresponding application.

The sender must still react to packet acknowledgments:

- When the sender gets an acknowledgment, it releases the copy of the corresponding packet from the buffer.
- If the sender reaches the time-out instead of receiving an acknowledgment, it resends the packet and restarts the timer.

Just as a protocol is implemented at network end nodes to support communication, protocols are also used across the network structure at the physical, data link, and network layers responsible primarily for packet transport, flow control, error handling, and other functions described next.

Basic Network Structure and Functions: Media and Form Factor, Packet Transport, Flow Control, and Error Handling

Once a packet is ready for transmission at its source, it is injected into the network using some dedicated hardware at the network interface. The hardware includes some transceiver circuits to drive the physical network media—either electrical or optical. The type of *media* and *form factor* depends largely on the interconnect distances over which certain signaling rates (e.g., transmission speed) should be sustainable. For centimeter or less distances on a chip or multichip module, typically the middle to upper copper metal layers can be used for interconnects at multi-Gbps signaling rates per line. A dozen or more layers of copper traces or tracks imprinted on circuit boards, midplanes, and backplanes can be used for Gbps differential-pair signaling rates at distances of about a meter or so. Category 5E unshielded twisted-pair copper wiring allows 0.25 Gbps transmission speed over distances of 100 meters. Coaxial copper cables can deliver 10 Mbps over kilometer distances. In these conductor lines, distance can usually be traded off for higher transmission speed, up to a certain point. Optical media enable faster transmission speeds at distances of kilometers. Multimode fiber supports 100 Mbps transmission rates over a few kilometers, and more expensive single-mode fiber supports Gbps transmission speeds over distances of several kilometers. Wavelength division multiplexing allows several times more bandwidth to be achieved in fiber (i.e., by a factor of the number of wavelengths used).

The hardware used to drive network links may also include some encoders to encode the signal in a format other than binary that is suitable for the given transport distance. Encoding techniques can use multiple voltage levels, redundancy, data and control rotation (e.g., 4b5b encoding), and/or a guaranteed minimum number of signal transitions per unit time to allow for clock recovery at the receiver. The signal is decoded at the receiver end, and the packet is stored in the corresponding buffer. All of these operations are performed at the network physical layer, the details of which are beyond the scope of this appendix. Fortunately, we do not need to worry about them. From the perspective of the data link and higher layers, the physical layer can be viewed as a long linear pipeline without staging in which signals propagate as waves through the network transmission medium. All of the above functions are generally referred to as *packet transport*.

Besides packet transport, the network hardware and software are jointly responsible at the data link and network protocol layers for ensuring reliable delivery of packets. These responsibilities include: (1) preventing the sender from sending packets at a faster rate than they can be processed by the receiver, and (2) ensuring that the packet is neither garbled nor lost in transit. The first responsibility is met by either discarding packets at the receiver when its buffer is full and later notifying the sender to retransmit them, or by notifying the sender to stop sending packets when the buffer becomes full and to resume later once it has room for more packets. The latter strategy is generally known as *flow control*.

There are several interesting techniques commonly used to implement flow control beyond simple *handshaking* between the sender and receiver. The more popular techniques are *Xon/Xoff* (also referred to as *Stop & Go*) and *credit-based* flow control. Xon/Xoff consists of the receiver notifying the sender either to stop or to resume sending packets once high and low buffer occupancy levels are reached, respectively, with some hysteresis to reduce the number of notifications. Notifications are sent as "stop" and "go" signals using additional control wires or encoded in control packets. Credit-based flow control typically uses a credit counter at the sender that initially contains a number of credits equal to the number of buffers at the receiver. Every time a packet is transmitted, the sender decrements the credit counter. When the receiver consumes a packet from its buffer, it returns a credit to the sender in the form of a control packet that notifies the sender to increment its counter upon receipt of the credit. These techniques essentially control the flow of packets into the network by *throttling* packet injection at the sender when the receiver reaches a low watermark or when the sender runs out of credits.

Xon/Xoff usually generates much less control traffic than credit-based flow control because notifications are only sent when the high or low buffer occupancy levels are crossed. On the other hand, credit-based flow control requires less than half the buffer size required by Xon/Xoff. Buffers for Xon/Xoff must be large enough to prevent overflow before the "stop" control signal reaches the sender. Overflow cannot happen when using credit-based flow control because the sender will run out of credits, thus stopping transmission. For both schemes, full link bandwidth utilization is possible only if buffers are large enough for the distance over which communication takes place.

Let's compare the buffering requirements of the two flow control techniques in a simple example covering the various interconnection network domains.

Example Suppose we have a dedicated-link network with a raw data bandwidth of 8 Gbps for each link in each direction interconnecting two devices. Packets of 100 bytes (including the header) are continuously transmitted from one device to the other to fully utilize network bandwidth. What is the minimum amount of credits and buffer space required by credit-based flow control assuming interconnect distances of 1 cm, 1 m, 100 m, and 10 km if only link propagation delay is taken into account? How does the minimum buffer space compare against Xon/Xoff?

Answer At the start, the receiver buffer is initially empty and the sender contains a number of credits equal to buffer capacity. The sender will consume a credit every time a packet is transmitted. For the sender to continue transmitting packets at network speed, the first returned credit must reach the sender before the sender runs out of credits. After receiving the first credit, the sender will keep receiving credits at the same rate it transmits packets. As we are considering only propagation delay over the link and no other sources of delay or overhead, null processing time at the sender and receiver are assumed. The time required for the first credit to reach the sender since it started transmission of the first packet is equal to the round-trip propagation delay for the packet transmitted to the receiver and the return credit transmitted back to the sender. This time must be less than or equal to the packet transmission time multiplied by the initial credit count:

$$\text{Packet propagation delay} + \text{Credit propagation delay} \leq \frac{\text{Packet size}}{\text{Bandwidth}} \times \text{Credit count}$$

The speed of light is about 300,000 km/sec. Assume we can achieve 66% of that in a conductor. Thus, the minimum number of credits for each distance is given by

$$\left(\frac{\text{Distance}}{2/3 \times 300{,}000\ \text{km/sec}}\right) \times 2 \leq \frac{100\ \text{bytes}}{8\ \text{Gbits/sec}} \times \text{Credit count}$$

As each credit represents one packet-sized buffer entry, the minimum amount of credits (and, likewise, buffer space) needed by each device is one for the 1 cm and 1 m distances, 10 for the 100 m distance, and 1000 packets for the 10 km distance. For Xon/Xoff, this minimum buffer size corresponds to the buffer fragment from the high occupancy level to the top of the buffer and from the low occupancy level to the bottom of the buffer. With the added hysteresis between both occupancy levels to reduce notifications, the minimum buffer space for Xon/Xoff turns out to be more than twice that for credit-based flow control.

Networks that implement flow control do not need to drop packets and are sometimes referred to as *lossless* networks; networks that drop packets are sometimes referred to as *lossy* networks. This single difference in the way packets are handled by the network drastically constrains the kinds of solutions that can be implemented to address other related network problems, including packet routing, congestion, deadlock, and reliability, as we will see later in this appendix. This difference also affects performance significantly as dropped packets need to be retransmitted, thus consuming more link bandwidth and suffering extra delay. These behavioral and performance differences ultimately restrict the interconnection network domains for which certain solutions are applicable. For instance, most networks delivering packets over relatively short distances (e.g., OCNs and SANs) tend to implement flow control; on the other hand, networks delivering packets over relatively long distances (e.g., LANs and WANs) tend to be designed to drop packets. For the shorter distances, the delay in propagating flow control information back to the sender can be negligible, but not so for longer distance scales. The kinds of applications that are usually run also influence the choice of lossless versus lossy networks. For instance, dropping packets sent by an Internet client like a Web browser affects only the delay observed by the corresponding user. However, dropping a packet sent by a process from a parallel application may lead to a significant increase in the overall execution time of the application if that packet's delay is on the critical path.

The second responsibility of ensuring that packets are neither garbled nor lost in transit can be met by implementing some mechanisms to detect and recover from transport errors. Adding a checksum or some other error detection field to the packet format, as shown in Figure F.4, allows the receiver to detect errors. This redundant information is calculated when the packet is sent and checked upon receipt. The receiver then sends an acknowledgment in the form of a control packet if the packet passes the test. Note that this acknowledgment control packet may simultaneously contain flow control information (e.g., a credit or stop signal), thus reducing control packet overhead. As described earlier, the most common way to recover from errors is to have a timer record the time each packet is sent and to presume the packet is lost or erroneously transported if the timer expires before an acknowledgment arrives. The packet is then resent.

The communication protocol across the network and network end nodes must handle many more issues other than packet transport, flow control, and reliability. For example, if two devices are from different manufacturers, they might order bytes differently within a word (Big Endian versus Little Endian byte ordering). The protocol must reverse the order of bytes in each word as part of the delivery system. It must also guard against the possibility of duplicate packets if a delayed packet were to become unstuck. Depending on the system requirements, the protocol may have to implement *pipelining* among operations to improve performance. Finally, the protocol may need to handle network congestion to prevent performance degradation when more than two devices are connected, as described later in Section F.7.

Characterizing Performance: Latency and Effective Bandwidth

Now that we have covered the basic steps for sending and receiving messages between two devices, we can discuss performance. We start by discussing the latency when transporting a single packet. Then we discuss the effective bandwidth (also known as throughput) that can be achieved when the transmission of multiple packets is pipelined over the network at the packet level.

Figure F.5 shows the basic components of latency for a single packet. Note that some latency components will be broken down further in later sections as the internals of the "black box" network are revealed. The timing parameters in Figure F.5 apply to many interconnection network domains: inside a chip, between chips on a board, between boards in a chassis, between chassis within a computer, between computers in a cluster, between clusters, and so on. The values may change, but the components of latency remain the same.

The following terms are often used loosely, leading to confusion, so we define them here more precisely:

- *Bandwidth*—Strictly speaking, the *bandwidth* of a transmission medium refers to the range of frequencies for which the attenuation per unit length introduced by that medium is below a certain threshold. It must be distinguished from the *transmission speed*, which is the amount of information transmitted over a medium per unit time. For example, modems successfully increased transmission speed in the late 1990s for a fixed bandwidth (i.e., the 3 KHz bandwidth provided by voice channels over telephone lines) by encoding more voltage levels and, hence, more bits per signal cycle. However, to be consistent with

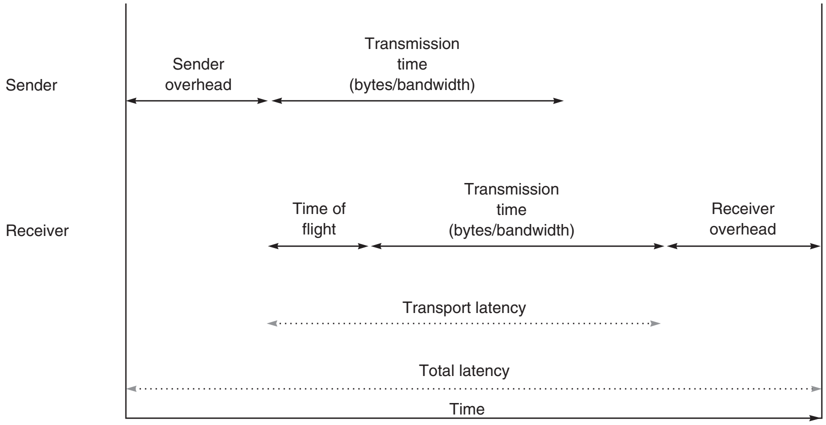


Figure F.5 Components of packet latency. Depending on whether it is an OCN, SAN, LAN, or WAN, the relative amounts of sending and receiving overhead, time of flight, and transmission time are usually quite different from those illustrated here.

its more widely understood meaning, we use the term *band-width* to refer to the maximum rate at which information can be transferred, where information includes packet header, payload, and trailer. The units are traditionally bits per second, although bytes per second is sometimes used. The term *bandwidth* is also used to mean the measured speed of the medium (i.e., network links). *Aggregate bandwidth* refers to the total data bandwidth supplied by the network, and *effective bandwidth* or *throughput* is the fraction of aggregate bandwidth delivered by the network to an application.

- *Time of flight*—This is the time for the first bit of the packet to arrive at the receiver, including the propagation delay over the links and delays due to other hardware in the network such as link repeaters and network switches. The unit of measure for time of flight can be in milliseconds for WANs, microseconds for LANs, nanoseconds for SANs, and picoseconds for OCNs.
- *Transmission time*—This is the time for the packet to pass through the network, not including time of flight. One way to measure it is the difference in time between when the first bit of the packet arrives at the receiver and when the last bit of that packet arrives at the receiver. By definition, transmission time is equal to the size of the packet divided by the data bandwidth of network links. This measure assumes there are no other packets contending for that bandwidth (i.e., a zero-load or no-load network).
- *Transport latency*—This is the sum of time of flight and transmission time. Transport latency is the time that the packet spends in the interconnection network. Stated alternatively, it is the time between when the first bit of the packet is injected into the network and when the last bit of that packet arrives at the receiver. It does not include the overhead of preparing the packet at the sender or processing it when it arrives at the receiver.
- *Sending overhead*—This is the time for the end node to prepare the packet (as opposed to the message) for injection into the network, including both hardware and software components. Note that the end node is busy for the entire time, hence the use of the term *overhead*. Once the end node is free, any subsequent delays are considered part of the transport latency. We assume that overhead consists of a constant term plus a variable term that depends on packet size. The constant term includes memory allocation, packet header preparation, setting up DMA devices, and so on. The variable term is mostly due to copies from buffer to buffer and is usually negligible for very short packets.
- *Receiving overhead*—This is the time for the end node to process an incoming packet, including both hardware and software components. We also assume here that overhead consists of a constant term plus a variable term that depends on packet size. In general, the receiving overhead is larger than the sending overhead. For example, the receiver may pay the cost of an interrupt or may have to reorder and reassemble packets into messages.

The total latency of a packet can be expressed algebraically by the following:

$$\text{Latency} = \text{Sending overhead} + \text{Time of flight} + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}$$

Let's see how the various components of transport latency and the sending and receiving overheads change in importance as we go across the interconnection network domains: from OCNs to SANs to LANs to WANs.

Example Assume that we have a dedicated link network with a data bandwidth of 8 Gbps for each link in each direction interconnecting two devices within an OCN, SAN, LAN, or WAN, and we wish to transmit packets of 100 bytes (including the header) between the devices. The end nodes have a per-packet sending overhead of $x + 0.05$ ns/byte and receiving overhead of $4/3(x) + 0.05$ ns/byte, where x is 0 μs for the OCN, 0.3 μs for the SAN, 3 μs for the LAN, and 30 μs for the WAN, which are typical for these network types. Calculate the total latency to send packets from one device to the other for interconnection distances of 0.5 cm, 5 m, 5000 m, and 5000 km assuming that time of flight consists only of link propagation delay (i.e., no switching or other sources of delay).

Answer Using the above expression and the calculation for propagation delay through a conductor given in the previous example, we can plug in the parameters for each of the networks to find their total packet latency. For the OCN:

$$\begin{aligned}\text{Latency} &= \text{Sending overhead} + \text{Time of flight} + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}\\ &= 5\text{ ns} + \frac{0.5\text{ cm}}{2/3 \times 300{,}000\text{ km/sec}} + \frac{100\text{ bytes}}{8\text{ Gbits/sec}} + 5\text{ ns}\end{aligned}$$

Converting all terms into nanoseconds (ns) leads to the following for the OCN:

$$\begin{aligned}\text{Total latency (OCN)} &= 5\text{ ns} + \frac{0.5\text{ cm}}{2/3 \times 300{,}000\text{ km/sec}} + \frac{100 \times 8}{8}\text{ns} + 5\text{ ns}\\ &= 5\text{ ns} + 0.025\text{ ns} + 100\text{ ns} + 5\text{ ns}\\ &= 110.025\text{ ns}\end{aligned}$$

Substituting in the appropriate values for the SAN gives the following latency:

$$\begin{aligned}\text{Total latency (SAN)} &= 0.305\ \mu\text{s} + \frac{5\text{ m}}{2/3 \times 300{,}000\text{ km/sec}} + \frac{100\text{ bytes}}{8\text{ Gbits/sec}} + 0.405\ \mu\text{s}\\ &= 0.305\ \mu\text{s} + 0.025\ \mu\text{s} + 0.1\ \mu\text{s} + 0.405\ \mu\text{s}\\ &= 0.835\ \mu\text{s}\end{aligned}$$

Substituting in the appropriate values for the LAN gives the following latency:

$$\begin{aligned}\text{Total latency (LAN)} &= 3.005\ \mu\text{s} + \frac{5\ \text{km}}{2/3 \times 300{,}000\ \text{km/sec}} + \frac{100\ \text{bytes}}{8\ \text{Gbits/sec}} + 4.005\ \mu\text{s}\\ &= 3.005\ \mu\text{s} + 25\ \mu\text{s} + 0.1\ \mu\text{s} + 4.005\ \mu\text{s}\\ &= 32.11\ \mu\text{s}\end{aligned}$$

Substituting in the appropriate values for the WAN gives the following latency:

$$\begin{aligned}\text{Total latency (WAN)} &= 30.005\ \mu\text{s} + \frac{5000\ \text{km}}{2/3 \times 300{,}000\ \text{km/sec}} + \frac{100\ \text{bytes}}{8\ \text{Gbits/sec}} + 40.005\ \mu\text{s}\\ &= 30.005\ \mu\text{s} + 25000\ \mu\text{s} + 0.1\ \mu\text{s} + 40.005\ \mu\text{s}\\ &= 25.07\ \text{ms}\end{aligned}$$

The increased fraction of the latency required by time of flight for the longer distances along with the greater likelihood of errors over the longer distances are among the reasons why WANs and LANs use more sophisticated and time-consuming communication protocols, which increase sending and receiving overheads. The need for standardization is another reason. Complexity also increases due to the requirements imposed on the protocol by the typical applications that run over the various interconnection network domains as we go from tens to hundreds to thousands to many thousands of devices. We will consider this in later sections when we discuss connecting more than two devices. The above example shows that the propagation delay component of time of flight for WANs and some LANs is so long that other latency components—including the sending and receiving overheads—can practically be ignored. This is not so for SANs and OCNs where the propagation delay pales in comparison to the overheads and transmission delay. Remember that time-of-flight latency due to switches and other hardware in the network besides sheer propagation delay through the links is neglected in the above example. For noncongested networks, switch latency generally is small compared to the overheads and propagation delay through the links in WANs and LANs, but this is not necessarily so for multiprocessor SANs and multicore OCNs, as we will see in later sections.

So far, we have considered the transport of a single packet and computed the associated end-to-end total packet latency. In order to compute the effective bandwidth for two networked devices, we have to consider a continuous stream of packets transported between them. We must keep in mind that, in addition to minimizing packet latency, the goal of any network optimized for a given cost and power consumption target is to transfer the maximum amount of available information in the shortest possible time, as measured by the effective bandwidth delivered by the network. For applications that do not require a response before sending the next packet, the sender can overlap the sending overhead of later packets with the transport latency and receiver overhead of prior packets. This essentially pipelines the transmission of packets over the network, also known as *link pipelining*. Fortunately, as discussed in prior chapters of this book, there are many application areas where communication from either several applications or several threads from the same application can run concurrently (e.g., a Web server concurrently serving thousands of client requests or streaming media), thus allowing a device to send a stream of packets without having to wait for an acknowledgment or a reply. Also, as long messages are usually divided into packets of maximum size before transport, a number of packets are injected into the network in succession for such cases. If such overlap were not possible, packets would have to wait for prior packets to be acknowledged before being transmitted and thus suffer significant performance degradation.

Packets transported in a pipelined fashion can be acknowledged quite straightforwardly simply by keeping a copy at the source of all unacknowledged packets that have been sent and keeping track of the correspondence between returned acknowledgments and packets stored in the buffer. Packets will be removed from the buffer when the corresponding acknowledgment is received by the sender. This can be done by including the message ID and packet sequence number associated with the packet in the packet's acknowledgment. Furthermore, a separate timer must be associated with each buffered packet, allowing the packet to be resent if the associated time-out expires.

Pipelining packet transport over the network has many similarities with pipelining computation within a processor. However, among some differences are that it does not require any staging latches. Information is simply propagated through network links as a sequence of signal waves. Thus, the network can be considered as a logical pipeline consisting of as many stages as are required so that the time of flight does not affect the effective bandwidth that can be achieved. Transmission of a packet can start immediately after the transmission of the previous one, thus overlapping the sending overhead of a packet with the transport and receiver latency of previous packets. If the sending overhead is smaller than the transmission time, packets follow each other back-to-back, and the effective bandwidth approaches the raw link bandwidth when continuously transmitting packets. On the other hand, if the sending overhead is greater than the transmission time, the effective bandwidth at the injection point will remain well below the raw link bandwidth. The resulting *link injection bandwidth*, $\mathrm{BW}_{\mathrm{LinkInjection}}$, for each link injecting a continuous stream of packets into a network is calculated with the following expression:

$$\mathrm{BW}_{\mathrm{LinkInjection}} = \frac{\text{Packet size}}{\max(\text{Sending overhead}, \text{Transmission time})}$$

We must also consider what happens if the receiver is unable to consume packets at the same rate they arrive. This occurs if the receiving overhead is greater than the sending overhead and the receiver cannot process incoming packets fast enough. In this case, the *link reception bandwidth*, $\mathrm{BW}_{\mathrm{LinkReception}}$, for each reception link of the network is less than the link injection bandwidth and is obtained with this expression:

$$\mathrm{BW}_{\mathrm{LinkReception}} = \frac{\text{Packet size}}{\max(\text{Receiving overhead}, \text{Transmission time})}$$

When communication takes place between two devices interconnected by dedicated links, all the packets sent by one device will be received by the other. If the receiver cannot process packets fast enough, the receiver buffer will become full, and flow control will throttle transmission at the sender. As this situation is produced by causes external to the network, we will not consider it further here. Moreover, if the receiving overhead is greater than the sending overhead, the receiver buffer will fill up and flow control will, likewise, throttle transmission at the sender. In this case, the effect of flow control is, on average, the same as if we replace sending overhead with receiving overhead. Assuming an ideal network that behaves like two dedicated links running in opposite directions at the full link bandwidth between the two devices—which is consistent with our black box view of the network to this point—the resulting effective bandwidth is the smaller of twice the injection bandwidth (to account for the two injection links, one for each device) or twice the reception bandwidth. This results in the following expression for effective bandwidth:

$$\text{Effective bandwidth} = \min\left(2 \times \text{BW}_{\text{LinkInjection}}, 2 \times \text{BW}_{\text{LinkReception}}\right) = \frac{2 \times \text{Packet size}}{\max\left(\text{Overhead}, \text{Transmission time}\right)}$$

where Overhead = max(Sending overhead, Receiving overhead). Taking into account the expression for the transmission time, it is obvious that the effective bandwidth delivered by the network is identical to the aggregate network bandwidth when the transmission time is greater than the overhead. Therefore, full network utilization is achieved regardless of the value for the time of flight and, thus, regardless of the distance traveled by packets, assuming ideal network behavior (i.e., enough credits and buffers are provided for credit-based and Xon/Xoff flow control). This analysis assumes that the sender and receiver network interfaces can process only one packet at a time. If multiple packets can be processed in parallel (e.g., as is done in IBM's Federation network interfaces), the overheads for those packets can be overlapped, which increases effective bandwidth by that overlap factor up to the amount bounded by the transmission time.

Let's use the equation on page F-17 to explore the impact of packet size, transmission time, and overhead on $\text{BW}_{\text{Link Injection}}$, $\text{BW}_{\text{LinkReception}}$, and effective bandwidth for the various network domains: OCNs, SANs, LANs, and WANs.

Example As in the previous example, assume we have a dedicated link network with a data bandwidth of 8 Gbps for each link in each direction interconnecting the two devices within an OCN, SAN, LAN, or WAN. Plot effective bandwidth versus packet size for each type of network for packets ranging in size from 4 bytes (i.e., a single 32-bit word) to 1500 bytes (i.e., the maximum transfer unit for Ethernet), assuming that end nodes have the same per-packet sending and receiving overheads as before: $x + 0.05$ ns/byte and $4/3(x) + 0.05$ ns/byte, respectively, where x is 0 μs for the OCN, 0.3 μs for the SAN, 3 μs for the LAN, and 30 μs for the WAN. What limits the effective bandwidth, and for what packet sizes is the effective bandwidth within 10% of the aggregate network bandwidth?

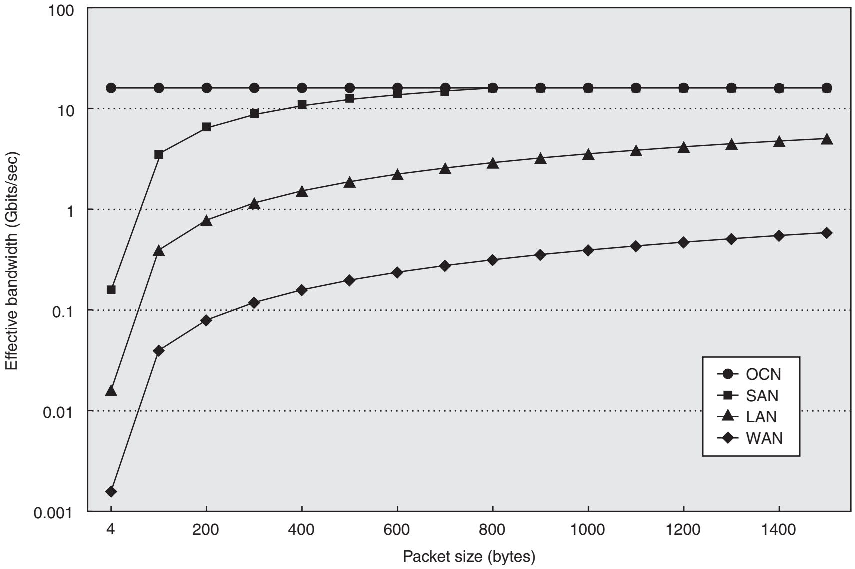


Figure F.6 Effective bandwidth versus packet size plotted in semi-log form for the four network domains. Overhead can be amortized by increasing the packet size, but for too large of an overhead (e.g., for WANs and some LANs) scaling the packet size is of little help. Other considerations come into play that limit the maximum packet size.

Answer Figure F.6 plots effective bandwidth versus packet size for the four network domains using the simple equation and parameters given above. For all packet sizes in the OCN, transmission time is greater than overhead (sending or receiving), allowing full utilization of the aggregate bandwidth, which is 16 Gbps—that is, injection link (alternatively, reception link) bandwidth times two to account for both devices. For the SAN, overhead—specifically, receiving overhead—is larger than transmission time for packets less than about 800 bytes; consequently, packets of 655 bytes and larger are needed to utilize 90% or more of the aggregate bandwidth. For LANs and WANs, most of the link bandwidth is not utilized since overhead in this example is many times larger than transmission time for all packet sizes.

This example highlights the importance of reducing the sending and receiving overheads relative to packet transmission time in order to maximize the effective bandwidth delivered by the network.

The analysis above suggests that it is possible to provide some upper bound for the effective bandwidth by analyzing the path followed by packets and determining where the bottleneck occurs. We can extend this idea beyond the network interfaces by defining a model that considers the entire network from end to end as a pipe and identifying the narrowest section of that pipe. There are three areas of interest in that pipe: the aggregate of all network injection links and the corresponding *network injection bandwidth* ($BW_{NetworkInjection}$), the aggregate of all network reception links and the corresponding *network reception bandwidth* ($BW_{NetworkReception}$), and the aggregate of all network links and the corresponding *network bandwidth* ($BW_{Network}$). Expressions for these will be given in later sections as various layers of the black box view of the network are peeled away.

To this point, we have assumed that for just two interconnected devices the black box network behaves ideally and the network bandwidth is equal to the aggregate raw network bandwidth. In reality, it can be much less than the aggregate bandwidth as we will see in the following sections. In general, the effective bandwidth delivered end-to-end by the network to an application is upper bounded by the minimum across all three potential bottleneck areas:

$$\text{Effective bandwidth} = \min\left(BW_{\text{NetworkInjection}}, BW_{\text{Network}}, BW_{\text{NetworkReception}}\right)$$

We will expand upon this expression further in the following sections as we reveal more about interconnection networks and consider the more general case of interconnecting more than two devices.

In some sections of this appendix, we show how the concepts introduced in the section take shape in example high-end commercial products. Figure F.7 lists several commercial computers that, at one point in time in their existence, were among the highest-performing systems in the world within their class. Although these systems are capable of interconnecting more than two devices, they implement the basic functions needed for interconnecting only two devices. In addition to being applicable to the SANs used in those systems, the issues discussed in this section also apply to other interconnect domains: from OCNs to WANs.

F.3 Connecting More than Two Devices

To this point, we have considered the connection of only two devices communicating over a network viewed as a black box, but what makes interconnection networks interesting is the ability to connect hundreds or even many thousands of devices together. Consequently, what makes them interesting also makes them more challenging to build. In order to connect more than two devices, a suitable structure and more functionality must be supported by the network. This section continues with our black box approach by introducing, at a conceptual level, additional network structure and functions that must be supported when interconnecting more than two devices. More details on these individual subjects are given in Sections F.4 through F.7. Where applicable, we relate the additional structure and functions to network media, flow control, and other basics presented in the previous section. In this section, we also classify networks into two broad categories

Company	System [network] name	Intro year	Max. number of compute nodes [# CPUs]	System footprint for max. configuration	Packet [header] max size (bytes)	Injection [reception] node BW in MB/sec	Minimum send/receive overhead	Maximum copper link length; flow control; error
Intel	ASCI Red Paragon	2001	4510 [×2]	2500 ft^2	1984 [4]	400 [400]	Few µs	Handshaking; CRC + parity
IBM	ASCI White SP Power3 [Colony]	2001	512 [×16]	10,000 ft^2	1024 [6]	500 [500]	$\sim$3 µs	25 m; credit-based; CRC
Intel	Thunder Itanium2 Tiger4 [QsNet^{II}]	2004	1024 [×4]	120 m^2	2048 [14]	928 [928]	0.240 µs	13 m; credit-based; CRC for link, dest.
Cray	XT3 [SeaStar]	2004	30,508 [×1]	263.8 m^2	80 [16]	3200 [3200]	Few µs	7 m; credit-based; CRC
Cray	X1E	2004	1024 [×1]	27 m^2	32 [16]	1600 [1600]	0 (direct LD ST accesses)	5 m; credit-based; CRC
IBM	ASC Purple pSeries 575 [Federation]	2005	>1280 [×8]	6720 ft^2	2048 [7]	2000 [2000]	$\sim$1 µs with up to 4 packets processed in $\|$	25 m; credit-based; CRC
IBM	Blue Gene/L eServer Sol. [Torus Net.]	2005	65,536 [×2]	2500 ft^2 (.9 × .9 × 1.9 m^3/ 1 K node rack)	256 [8]	612.5 [1050]	$\sim$3 µs (2300 cycles)	8.6 m; credit-based; CRC (header/pkt)

Figure F.7 Basic characteristics of interconnection networks in commercial high-performance computer systems.

based on their connection structure—*shared-media* versus *switched-media* networks—and we compare them. Finally, expanded expressions for characterizing network performance are given, followed by an example.

Additional Network Structure and Functions: Topology, Routing, Arbitration, and Switching

Networks interconnecting more than two devices require mechanisms to physically connect the packet source to its destination in order to transport the packet and deliver it to the correct destination. These mechanisms can be implemented in different ways and significantly vary across interconnection network domains. However, the types of network structure and functions performed by those mechanisms are very much the same, regardless of the domain.

When multiple devices are interconnected by a network, the connections between them oftentimes cannot be permanently established with dedicated links.

This could either be too restrictive as all the packets from a given source would go to the same one destination (and not to others) or prohibitively expensive as a dedicated link would be needed from every source to every destination (we will evaluate this further in the next section). Therefore, networks usually share paths among different pairs of devices, but how those paths are shared is determined by the network connection structure, commonly referred to as the network *topology*. Topology addresses the important issue of "*What paths are possible for packets?*" so packets reach their intended destinations.

Every network that interconnects more than two devices also requires some mechanism to deliver each packet to the correct destination. The associated function is referred to as *routing*, which can be defined as the set of operations that need to be performed to compute a valid path from the packet source to its destinations. Routing addresses the important issue of "*Which of the possible paths are allowable (valid) for packets?*" so packets reach their intended destinations. Depending on the network, this function may be executed at the packet source to compute the entire path, at some intermediate devices to compute fragments of the path on the fly, or even at every possible destination device to verify whether that device is the intended destination for the packet. Usually, the packet header shown in Figure F.4 is extended to include the necessary routing information.

In general, as networks usually contain shared paths or parts thereof among different pairs of devices, packets may request some shared resources. When several packets request the same resources at the same time, an *arbitration* function is required to resolve the conflict. Arbitration, along with flow control, addresses the important issue of "*When are paths available for packets?*" Every time arbitration is performed, there is a winner and possibly several losers. The losers are not granted access to the requested resources and are typically buffered. As indicated in the previous section, flow control may be implemented to prevent buffer overflow. The winner proceeds toward its destination once the granted resources are switched in, providing a path for the packet to advance. This function is referred to as *switching*. Switching addresses the important issue of "*How are paths allocated to packets?*" To achieve better utilization of existing communication resources, most networks do not establish an entire end-to-end path at once. Instead, as explained in Section F.5, paths are usually established one fragment at a time.

These three network functions—routing, arbitration, and switching—must be implemented in every network connecting more than two devices, no matter what form the network topology takes. This is in addition to the basic functions mentioned in the previous section. However, the complexity of these functions and the order in which they are performed depends on the category of network topology, as discussed below. In general, routing, arbitration, and switching are required to establish a valid path from source to destination from among the possible paths provided by the network topology. Once the path has been established, the packet transport functions previously described are used to reliably transmit packets and receive them at the corresponding destination. Flow control, if implemented, prevents buffer overflow by throttling the sender. It can be implemented at the end-to-end level, the link level within the network, or both.

Shared-Media Networks

The simplest way to connect multiple devices is to have them share the network media, as shown for the bus in Figure F.8 (a). This has been the traditional way of interconnecting devices. The shared media can operate in *half-duplex* mode, where data can be carried in either direction over the media but simultaneous transmission and reception of data by the same device is not allowed, or in *full-duplex,* where the data can be carried in both directions and simultaneously transmitted and received by the same device. Until very recently, I/O devices in most systems typically shared a single I/O bus, and early system-on-chip (SoC) designs made use of a shared bus to interconnect on-chip components. The most popular LAN, Ethernet, was originally implemented as a half-duplex bus shared by up to a hundred computers, although now switched-media versions also exist.

Given that network media are shared, there must be a mechanism to coordinate and arbitrate the use of the shared media so that only one packet is sent at a time. If the physical distance between network devices is small, it may be possible to have a central arbiter to grant permission to send packets. In this case, the network nodes may use dedicated control lines to interface with the arbiter. Centralized arbitration is impractical, however, for networks with a large number of nodes spread over large distances, so distributed forms of arbitration are also used. This is the case for the original Ethernet shared-media LAN.

A first step toward distributed arbitration of shared media is "looking before you leap." A node first checks the network to avoid trying to send a packet while another packet is already in the network. Listening before transmission to avoid collisions is called *carrier sensing*. If the interconnection is idle, the node tries to send. Looking first is not a guarantee of success, of course, as some other node may also decide to send at the same instant. When two nodes send at the same time,

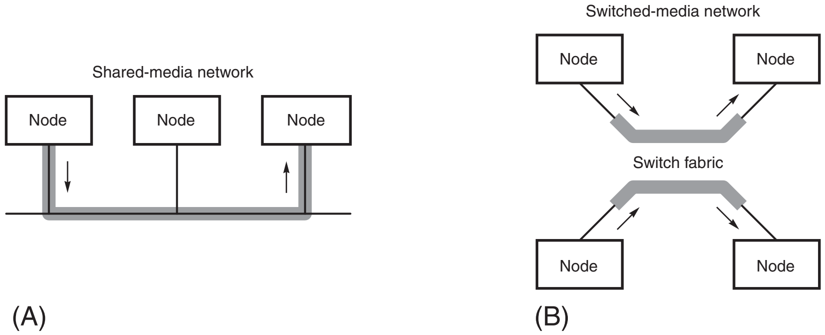


Figure F.8 (a) A shared-media network versus (b) a switched-media network. Ethernet was originally a shared media network, but switched Ethernet is now available. All nodes on the shared-media networks must dynamically share the raw bandwidth of one link, but switched-media networks can support multiple links, providing higher raw aggregate bandwidth.

a *collision* occurs. Let's assume that the network interface can detect any resulting collisions by listening to hear if the data become garbled by other data appearing on the line. Listening to detect collisions is called *collision detection*. This is the second step of distributed arbitration.

The problem is not solved yet. If, after detecting a collision, every node on the network waited exactly the same amount of time, listened to be sure there was no traffic, and then tried to send again, we could still have synchronized nodes that would repeatedly bump heads. To avoid repeated head-on collisions, each node whose packet gets garbled waits (or *backs off*) a random amount of time before resending. Randomization breaks the synchronization. Subsequent collisions result in exponentially increasing time between attempts to retransmit, so as not to tax the network.

Although this approach controls congestion on the shared media, it is not guaranteed to be fair—some subsequent node may transmit while those that collided are waiting. If the network does not have high demand from many nodes, this simple approach works well. Under high utilization, however, performance degrades since the media are shared and fairness is not ensured. Another distributed approach to arbitration of shared media that can support fairness is to pass a token between nodes. The function of the token is to grant the acquiring node the right to use the network. If the token circulates in a cyclic fashion between the nodes, a certain amount of fairness is ensured in the arbitration process.

Once arbitration has been performed and a device has been granted access to the shared media, the function of switching is straightforward. The granted device simply needs to connect itself to the shared media, thus establishing a path to every possible destination. Also, routing is very simple to implement. Given that the media are shared and attached to all the devices, every device will see every packet. Therefore, each device just needs to check whether or not a given packet is intended for that device. A beneficial side effect of this strategy is that a device can send a packet to all the devices attached to the shared media through a single transmission. This style of communication is called *broadcasting*, in contrast to *unicasting,* in which each packet is intended for only one device. The shared media make it easy to broadcast a packet to every device or, alternatively, to a subset of devices, called *multicasting*.

Switched-Media Networks

The alternative to sharing the entire network media at once across all attached nodes is to switch between disjoint portions of it shared by the nodes. Those portions consist of passive *point-to-point links* between active *switch* components that dynamically establish communication between sets of source-destination pairs. These passive and active components make up what is referred to as the network *switch fabric* or *network fabric*, to which end nodes are connected. This approach is shown conceptually in Figure F.8(b). The switch fabric is described in greater detail in Sections F.4 through F.7, where various black box layers for switched-media networks are further revealed. Nevertheless, the high-level view shown in Figure F.8(b) illustrates the potential bandwidth improvement of switched-media networks over shared-media networks: aggregate bandwidth can be many times higher than that of shared-media networks, allowing the possibility of greater effective bandwidth to be achieved. At best, only one node at a time can transmit packets over the shared media, whereas it is possible for all attached nodes to do so over the switched-media network.

Like their shared-media counterparts, switched-media networks must implement the three additional functions previously mentioned: routing, arbitration, and switching. Every time a packet enters the network, it is routed in order to select a path toward its destination provided by the topology. The path requested by the packet must be granted by some centralized or distributed arbiter, which resolves conflicts among concurrent requests for resources along the same path. Once the requested resources are granted, the network "switches in" the required connections to establish the path and allows the packet to be forwarded toward its destination. If the requested resources are not granted, the packet is usually buffered, as mentioned previously. Routing, arbitration, and switching functions are usually performed within switched networks in this order, whereas in shared-media networks routing typically is the last function performed.

Comparison of Shared- and Switched-Media Networks

In general, the advantage of shared-media networks is their low cost, but, consequently, their aggregate network bandwidth does not scale at all with the number of interconnected devices. Also, a global arbitration scheme is required to resolve conflicting demands, possibly introducing another type of bottleneck and again limiting scalability. Moreover, every device attached to the shared media increases the parasitic capacitance of the electrical conductors, thus increasing the time of flight propagation delay accordingly and, possibly, clock cycle time. In addition, it is more difficult to pipeline packet transmission over the network as the shared media are continuously granted to different requesting devices.

The main advantage of switched-media networks is that the amount of network resources implemented scales with the number of connected devices, increasing the aggregate network bandwidth. These networks allow multiple pairs of nodes to communicate simultaneously, allowing much higher effective network bandwidth than that provided by shared-media networks. Also, switched-media networks allow the system to scale to very large numbers of nodes, which is not feasible when using shared media. Consequently, this scaling advantage can, at the same time, be a disadvantage if network resources grow superlinearly. Networks of superlinear cost that provide an effective network bandwidth that grows only sublinearly with the number of interconnected devices are inefficient designs for many applications and interconnection network domains.

Characterizing Performance: Latency and Effective Bandwidth

The routing, switching, and arbitration functionality described above introduces some additional components of packet transport latency that must be taken into account in the expression for total packet latency. Assuming there is no contention for network resources—as would be the case in an unloaded network—total packet latency is given by the following:

$$\text{Latency} = \text{Sending overhead} + \left(T_{\text{TotalProp}} + T_{\text{R}} + T_{\text{A}} + T_{\text{S}}\right) + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}$$

Here T_{R}, T_{A}, and T_{S} are the total routing time, arbitration time, and switching time experienced by the packet, respectively, and are either measured quantities or calculated quantities derived from more detailed analyses. These components are added to the total propagation delay through the network links, $T_{\text{TotalProp}}$, to give the overall time of flight of the packet.

The expression above gives only a lower bound for the total packet latency as it does not account for additional delays due to contention for resources that may occur. When the network is heavily loaded, several packets may request the same network resources concurrently, thus causing contention that degrades performance. Packets that lose arbitration have to be buffered, which increases packet latency by some *contention delay* amount of waiting time. This additional delay is not included in the above expression. When the network or part of it approaches saturation, contention delay may be several orders of magnitude greater than the total packet latency suffered by a packet under zero load or even under slightly loaded network conditions. Unfortunately, it is not easy to compute analytically the total packet latency when the network is more than moderately loaded. Measurement of these quantities using cycle-accurate simulation of a detailed network model is a better and more precise way of estimating packet latency under such circumstances. Nevertheless, the expression given above is useful in calculating *best-case lower bounds* for packet latency.

For similar reasons, effective bandwidth is not easy to compute exactly, but we can estimate *best-case upper bounds* for it by appropriately extending the model presented at the end of the previous section. What we need to do is to find the narrowest section of the end-to-end network pipe by finding the network injection bandwidth ($\text{BW}_{\text{NetworkInjection}}$), the network reception bandwidth ($\text{BW}_{\text{NetworkReception}}$), and the network bandwidth ($\text{BW}_{\text{Network}}$) across the entire network interconnecting the devices.

The $\text{BW}_{\text{NetworkInjection}}$ can be calculated simply by multiplying the expression for link injection bandwidth, $\text{BW}_{\text{LinkInjection}}$, by the total number of network injection links. The $\text{BW}_{\text{NetworkReception}}$ is calculated similarly using $\text{BW}_{\text{LinkReception}}$, but it must also be scaled by a factor that reflects application traffic and other characteristics. For more than two interconnected devices, it is no longer valid to assume a one-to-one relationship among sources and destinations when analyzing the effect of flow control on link reception bandwidth. It could happen, for example, that several packets from different injection links arrive concurrently at the same reception link for applications that have many-to-one traffic characteristics, which causes contention at the reception links. This effect can be taken into account by an *average reception factor* parameter, σ, which is either a measured quantity or a calculated quantity derived from detailed analysis. It is defined as the average fraction or percentage of packets arriving at reception links that can be accepted. Only those packets can be immediately delivered, thus reducing network reception bandwidth by that factor. This reduction occurs as a result of application behavior regardless of internal network characteristics. Finally, $\text{BW}_{\text{Network}}$ takes into account the internal characteristics of the network, including contention. We will progressively derive expressions in the following sections that will enable us to calculate this as more details are revealed about the internals of our black box interconnection network.

Overall, the effective bandwidth delivered by the network end-to-end to an application is determined by the minimum across the three sections, as described by the following:

$$\begin{aligned}\text{Effective bandwidth} &= \min\left(\text{BW}_{\text{NetworkInjection}}, \text{BW}_{\text{Network}}, \sigma \times \text{BW}_{\text{NetworkReception}}\right)\\ &= \min\left(\text{N} \times \text{BW}_{\text{LinkInjection}}, \text{BW}_{\text{Network}}, \sigma \times \text{N} \times \text{BW}_{\text{LinkReception}}\right)\end{aligned}$$

Let's use the above expressions to compare the latency and effective bandwidth of shared-media networks against switched-media networks for the four interconnection network domains: OCNs, SANs, LANs, and WANs.

Example Plot the total packet latency and effective bandwidth as the number of interconnected nodes, N, scales from 4 to 1024 for shared-media and switched-media OCNs, SANs, LANs, and WANs. Assume that all network links, including the injection and reception links at the nodes, each have a data bandwidth of 8 Gbps, and unicast packets of 100 bytes are transmitted. Shared-media networks share one link, and switched-media networks have at least as many network links as there are nodes. For both, ignore latency and bandwidth effects due to contention within the network. End nodes have per-packet sending and receiving overheads of $x + 0.05$ ns/byte and $4/3(x) + 0.05$ ns/byte, respectively, where x is 0 μs for the OCN, 0.3 μs for the SAN, 3 μs for the LAN, and 30 μs for the WAN, and interconnection distances are 0.5 cm, 5 m, 5000 m, and 5000 km, respectively. Also assume that the total routing, arbitration, and switching times are constants or functions of the number of interconnected nodes: $T_{\text{R}} = 2.5$ ns, $T_{\text{A}} = 2.5(N)$ ns, and $T_{\text{S}} = 2.5$ ns for shared-media networks and $T_{\text{R}} = T_{\text{A}} = T_{\text{S}} = 2.5(\log_2 N)$ ns for switched-media networks. Finally, taking into account application traffic characteristics for the network structure, the average reception factor, σ, is assumed to be N^{-1} for shared media and polylogarithmic $(\log_2 N)^{-1/4}$ for switched media.

Answer All components of total packet latency are the same as in the example given in the previous section except for time of flight, which now has additional routing, arbitration, and switching delays. For shared-media networks, the additional delays total $5 + 2.5(N)$ ns; for switched-media networks, they total $7.5(\log_2 N)$ ns. Latency is plotted only for OCNs and SANs in Figure F.9 as these networks give the more interesting results. For OCNs, T_{R}, T_{A}, and T_{S} combine to dominate time of flight

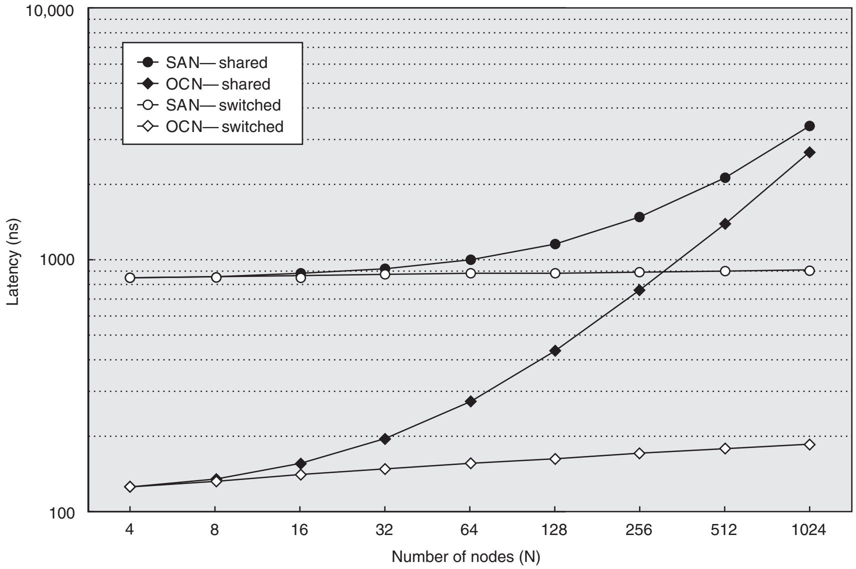


Figure F.9 Latency versus number of interconnected nodes plotted in semi-log form for OCNs and SANs. Routing, arbitration, and switching have more of an impact on latency for networks in these two domains, particularly for networks with a large number of nodes, given the low sending and receiving overheads and low propagation delay.

and are much greater than each of the other latency components for a moderate to large number of nodes. This is particularly so for the shared-media network. The latency increases much more dramatically with the number of nodes for shared media as compared to switched media given the difference in arbitration delay between the two. For SANs, T_R, T_A, and T_S dominate time of flight for most network sizes but are greater than each of the other latency components in shared-media networks only for large-sized networks; they are less than the other latency components for switched-media networks but are not negligible. For LANs and WANs, time of flight is dominated by propagation delay, which dominates other latency components as calculated in the previous section; thus, T_R, T_A, and T_S are negligible for both shared and switched media.

Figure F.10 plots effective bandwidth versus number of interconnected nodes for the four network domains. The effective bandwidth for all shared-media networks is constant through network scaling as only one unicast packet can be received at a time over all the network reception links, and that is further limited by the receiving overhead of each network for all but the OCN. The effective bandwidth for all switched-media networks increases with the number of interconnected nodes, but it is scaled down by the average reception factor. The receiving overhead further limits effective bandwidth for all but the OCN.

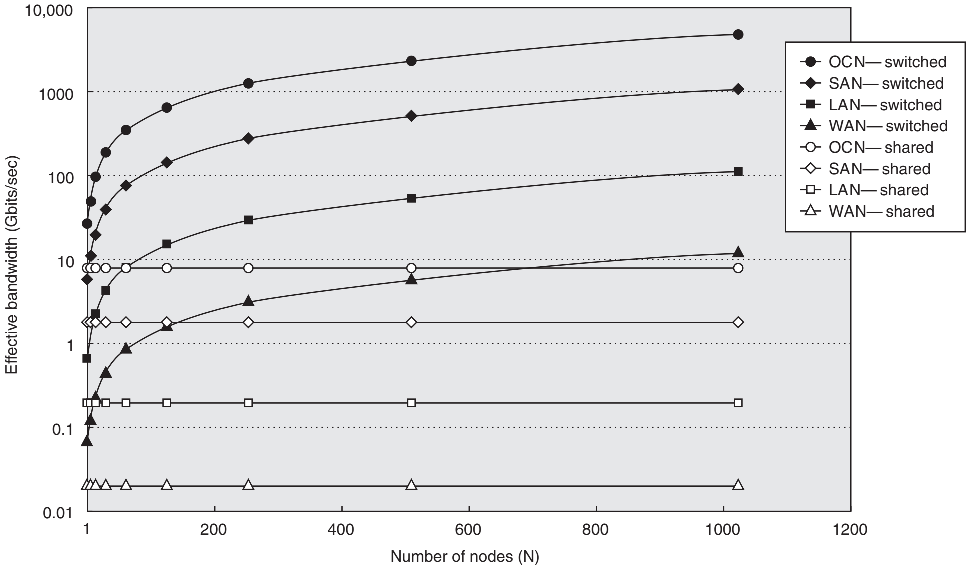


Figure F.10 Effective bandwidth versus number of interconnected nodes plotted in semi-log form for the four network domains. The disparity in effective bandwidth between shared- and switched-media networks for all interconnect domains widens significantly as the number of nodes in the network increases. Only the switched on-chip network is able to achieve an effective bandwidth equal to the aggregate bandwidth for the parameters given in this example.

Given the obvious advantages, why weren't switched networks always used? Earlier computers were much slower and could share the network media with little impact on performance. In addition, the switches for earlier LANs and WANs took up several large boards and were about as large as an entire computer. As a consequence of Moore's law, the size of switches has reduced considerably, and systems have a much greater need for high-performance communication. Switched networks allow communication to harvest the same rapid advancements from silicon as processors and main memory. Whereas switches from telecommunication companies were once the size of mainframe computers, today we see single-chip switches and even entire switched networks within a chip. Thus, technology and application trends favor switched networks today. Just as single-chip processors led to processors replacing logic circuits in a surprising number of places, single-chip switches and switched on-chip networks are increasingly replacing shared-media networks (i.e., buses) in several application domains. As an example, PCI-Express (PCIe)—a switched network—was introduced in 2005 to replace the traditional PCI-X bus on personal computer motherboards.

The previous example also highlights the importance of optimizing the routing, arbitration, and switching functions in OCNs and SANs. For these network domains in particular, the interconnect distances and overheads typically are small enough to make latency and effective bandwidth much more sensitive to how well these functions are implemented, particularly for larger-sized networks. This leads mostly to implementations based mainly on the faster hardware solutions for these domains. In LANs and WANs, implementations based on the slower but more flexible software solutions suffice given that performance is largely determined by other factors. The design of the topology for switched-media networks also plays a major role in determining how close to the lower bound on latency and the upper bound on effective bandwidth the network can achieve for OCN and SAN domains.

The next three sections touch on these important issues in switched networks, with the next section focused on topology.

F.4 Network Topology

When the number of devices is small enough, a single switch is sufficient to interconnect them within a switched-media network. However, the number of *switch ports* is limited by existing very-large-scale integration (VLSI) technology, cost considerations, power consumption, and so on. When the number of required *network ports* exceeds the number of ports supported by a single switch, a fabric of interconnected switches is needed. To embody the necessary property of *full access* (i.e., *connectedness*), the network switch fabric must provide a path from every end node device to every other device. All the connections to the network fabric and between switches within the fabric use point-to-point links as opposed to shared links—that is, links with only one switch or end node device on either end. The interconnection structure across all the components—including switches, links, and end node devices—is referred to as the network *topology*.

The number of network topologies described in the literature would be difficult to count, but the number that have been used commercially is no more than about a dozen or so. During the 1970s and early 1980s, researchers struggled to propose new topologies that could reduce the number of switches through which packets must traverse, referred to as the *hop count*. In the 1990s, thanks to the introduction of pipelined transmission and switching techniques, the hop count became less critical. Nevertheless, today, topology is still important, particularly for OCNs and SANs, as subtle relationships exist between topology and other network design parameters that impact performance, especially when the number of end nodes is very large (e.g., 64 K in the Blue Gene/L supercomputer) or when the latency is critical (e.g., in multicore processor chips). Topology also greatly impacts the implementation cost of the network.

Topologies for parallel supercomputer SANs have been the most visible and imaginative, usually converging on regularly structured ones to simplify routing, packaging, and scalability. Those for LANs and WANs tend to be more haphazard or ad hoc, having more to do with the challenges of long distance or connecting across different communication subnets. Switch-based topologies for OCNs are only recently emerging but are quickly gaining in popularity. This section describes the more popular topologies used in commercial products. Their advantages, disadvantages, and constraints are also briefly discussed.

Centralized Switched Networks

As mentioned above, a single switch suffices to interconnect a set of devices when the number of switch ports is equal to or larger than the number of devices. This simple network is usually referred to as a *crossbar* or *crossbar switch*. Within the crossbar, crosspoint switch complexity increases quadratically with the number of ports, as illustrated in Figure F.11(a). Thus, a cheaper solution is desirable when the number of devices to be interconnected scales beyond the point supportable by implementation technology.

A common way of addressing the crossbar scaling problem consists of splitting the large crossbar switch into several stages of smaller switches interconnected in such a way that a single pass through the switch fabric allows any destination to be reached from any source. Topologies arranged in this way are usually referred to as *multistage interconnection networks* or *multistage switch fabrics*, and these networks typically have complexity that increases in proportion to $N \log N$. Multistage interconnection networks (MINs) were initially proposed for telephone exchanges in the 1950s and have since been used to build the communication backbone for parallel supercomputers, symmetric multiprocessors, multicomputer clusters, and IP router switch fabrics.

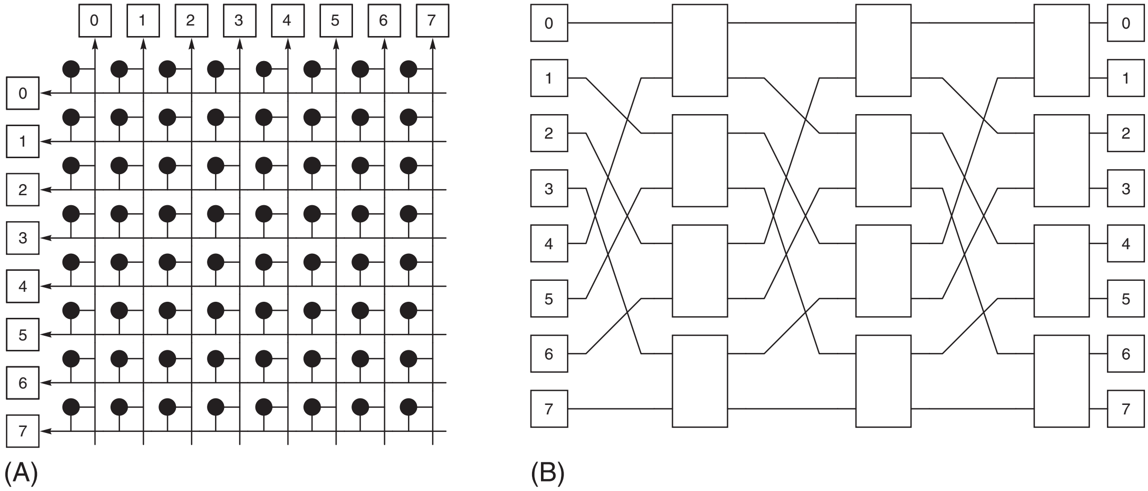


Figure F.11 Popular centralized switched networks: (a) the crossbar network requires N^2 crosspoint switches, shown as black dots; (b) the Omega, a MIN, requires $N/2 \log_2 N$ switches, shown as vertical rectangles. End node devices are shown as numbered squares (total of eight). Links are unidirectional—data enter at the left and exit out the top or right.

The interconnection pattern or patterns between MIN stages are permutations that can be represented mathematically by a set of functions, one for each stage. Figure F.11(b) shows a well-known MIN topology, the *Omega*, which uses the perfect-shuffle permutation as its interconnection pattern for each stage, followed by exchange switches, giving rise to a *perfect-shuffle exchange* for each stage. In this example, eight input-output ports are interconnected with three stages of 2 × 2 switches. It is easy to see that a single pass through the three stages allows any input port to reach any output port. In general, when using $k \times k$ switches, a MIN with N input-output ports requires at least $\log_k N$ stages, each of which contains N/k switches, for a total of $N/k\ (\log_k N)$ switches.

Despite their internal structure, MINs can be seen as centralized switch fabrics that have end node devices connected at the network periphery, hence the name *centralized switched network*. From another perspective, MINs can be viewed as interconnecting nodes through a set of switches that may not have any nodes directly connected to them, which gives rise to another popular name for centralized switched networks—*indirect networks*.

Example Compute the cost of interconnecting 4096 nodes using a single crossbar switch relative to doing so using a MIN built from 2 × 2, 4 × 4, and 16 × 16 switches. Consider separately the relative cost of the unidirectional links and the relative cost of the switches. Switch cost is assumed to grow quadratically with the number of input (alternatively, output) ports, k, for $k \times k$ switches.

Answer The switch cost of the network when using a single crossbar is proportional to 4096^2. The unidirectional link cost is 8192, which accounts for the set of links from the end nodes to the crossbar and also from the crossbar back to the end nodes. When using a MIN with $k \times k$ switches, the cost of each switch is proportional to k^2 but there are $4096/k\ (\log_k 4096)$ total switches. Likewise, there are $(\log_k 4096)$ stages of N unidirectional links per stage from the switches plus N links to the MIN from the end nodes. Therefore, the relative costs of the crossbar with respect to each MIN is given by the following:

$$\text{Relative cost } (2\times 2)_{switches} = 4096^2/\left(2^2 \times 4096/2 \times \log_2 4096\right) = 170$$

$$\text{Relative cost } (4\times 4)_{switches} = 4096^2/\left(4^2 \times 4096/4 \times \log_4 4096\right) = 170$$

$$\text{Relative cost } (16\times 16)_{switches} = 4096^2/\left(16^2 \times 4096/16 \times \log_{16} 4096\right) = 85$$

$$\text{Relative cost } (2\times 2)_{links} = 8192/(4096 \times (\log_2 4096 + 1)) = 2/13 = 0.1538$$

$$\text{Relative cost } (4\times 4)_{links} = 8192/(4096 \times (\log_4 4096 + 1)) = 2/7 = 0.2857$$

$$\text{Relative cost } (16\times 16)_{links} = 8192/(4096 \times (\log_{16} 4096 + 1)) = 2/4 = 0.5$$

In all cases, the single crossbar has much higher switch cost than the MINs. The most dramatic reduction in cost comes from the MIN composed from the smallest sized but largest number of switches, but it is interesting to see that the MINs with 2 × 2 and 4 × 4 switches yield the same relative switch cost. The relative link cost of the crossbar is lower than the MINs, but by less than an order of magnitude in all cases. We must keep in mind that end node links are different from switch links in their length and packaging requirements, so they usually have different associated costs. Despite the lower link cost, the crossbar has higher overall relative cost.

The reduction in switch cost of MINs comes at the price of performance: contention is more likely to occur on network links, thus degrading performance. Contention in the form of packets *blocking* in the network arises due to paths from different sources to different destinations simultaneously sharing one or more links. The amount of contention in the network depends on communication traffic behavior. In the Omega network shown in Figure F.11(b), for example, a packet from port 0 to port 1 blocks in the first stage of switches while waiting for a packet from port 4 to port 0. In the crossbar, no such blocking occurs as links are not shared among paths to unique destinations. The crossbar, therefore, is *nonblocking*. Of course, if two nodes try to send packets to the same destination, there will be blocking at the reception link even for crossbar networks. This is accounted for by the average reception factor parameter (σ) when analyzing performance, as discussed at the end of the previous section.

To reduce blocking in MINs, extra switches must be added or larger ones need to be used to provide alternative paths from every source to every destination. The first commonly used solution is to add a minimum of $\log_k N - 1$ extra switch stages to the MIN in such a way that they mirror the original topology. The resulting network is *rearrangeably nonblocking* as it allows nonconflicting paths among new source-destination pairs to be established, but it also doubles the hop count and could require the paths of some existing communicating pairs to be rearranged under some centralized control. The second solution takes a different approach. Instead of using more switch stages, larger switches—which can be implemented by multiple stages if desired—are used in the middle of two other switch stages in such a way that enough alternative paths through the middle-stage switches allow for nonconflicting paths to be established between the first and last stages. The best-known example of this is the Clos network, which is nonblocking. The multipath property of the three-stage Clos topology can be recursively applied to the middle-stage switches to reduce the size of all the switches down to 2×2, assuming that switches of this size are used in the first and last stages to begin with. What results is a Beneŝ topology consisting of $2(\log_2 N) - 1$ stages, which is rearrangeably nonblocking. Figure F.12(a) illustrates both topologies, where all switches not in the first and last stages comprise the middle-stage switches (recursively) of the Clos network.

The MINs described so far have unidirectional network links, but bidirectional forms are easily derived from symmetric networks such as the Clos and Beneŝ simply by folding them. The overlapping unidirectional links run in different directions, thus forming bidirectional links, and the overlapping switches merge into a single switch with twice the ports (i.e., 4×4 switch). Figure F.12(b) shows the resulting folded Beneŝ topology but in this case with the end nodes connected

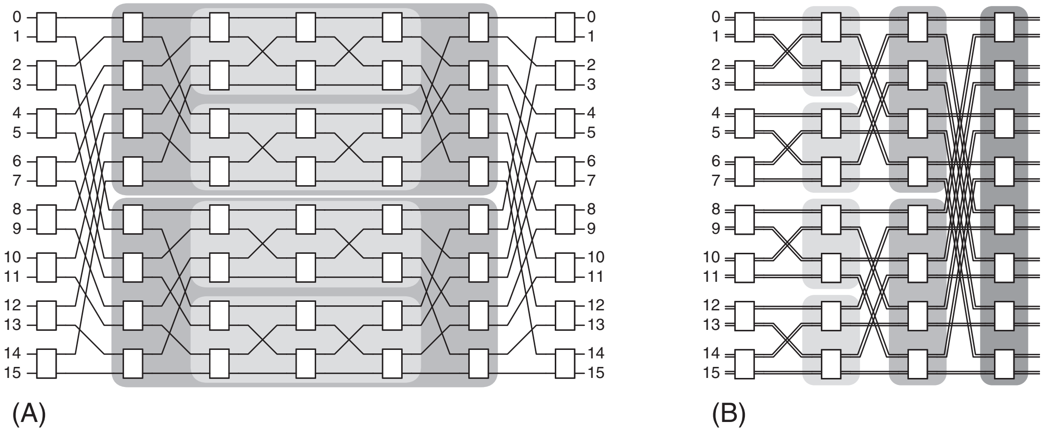


Figure F.12 Two Beneŝ networks. (a) A 16-port Clos topology, where the middle-stage switches shown in the darker shading are implemented with another Clos network whose middle-stage switches shown in the lighter shading are implemented with yet another Clos network, and so on, until a Beneŝ network is produced that uses only 2 × 2 switches everywhere. (b) A folded Beneŝ network (bidirectional) in which 4 × 4 switches are used; end nodes attach to the innermost set of the Beneŝ network (unidirectional) switches. This topology is equivalent to a fat tree, where tree vertices are shown in shades.

to the innermost switch stage of the original Beneŝ. Ports remain free at the other side of the network but can be used for later expansion of the network to larger sizes. These kind of networks are referred to as *bidirectional multistage interconnection networks.* Among many useful properties of these networks are their modularity and their ability to exploit communication locality, which saves packets from having to hop across all network stages. Their regularity also reduces routing complexity and their multipath property enables traffic to be routed more evenly across network resources and to tolerate faults.

Another way of deriving bidirectional MINs with nonblocking (rearrangeable) properties is to form a balanced tree, where end node devices occupy leaves of the tree and switches occupy vertices within the tree. Enough links in each tree level must be provided such that the total link bandwidth remains constant across all levels. Also, except for the root, switch ports for each vertex typically grow as $k^i \times k^i$, where i is the tree level. This can be accomplished by using k^{i-1} total switches at each vertex, where each switch has k input and k output ports, or k bidirectional ports (i.e., $k \times k$ input-output ports). Networks having such topologies are called *fat tree* networks. As only half of the k bidirectional ports are used in each direction, $2\ N/k$ switches are needed in each stage, totaling $2\ N/k\ (\log_{k/2} N)$ switches in the fat tree. The number of switches in the root stage can be halved as no forward links are needed, reducing switch count by N/k. Figure F.12(b) shows a fat tree for 4 × 4 switches. As can be seen, this is identical to the folded Beneŝ.

The fat tree is the topology of choice across a wide range of network sizes for most commercial systems that use multistage interconnection networks. Most SANs used in multicomputer clusters, and many used in the most powerful supercomputers, are based on fat trees. Commercial communication subsystems offered by Myrinet, Mellanox, and Quadrics are also built from fat trees.

Distributed Switched Networks

Switched-media networks provide a very flexible framework to design communication subsystems external to the devices that need to communicate, as presented above. However, there are cases where it is convenient to more tightly integrate the end node devices with the network resources used to enable them to communicate. Instead of centralizing the switch fabric in an external subsystem, an alternative approach is to distribute the network switches among the end nodes, which then become *network nodes* or simply *nodes*, yielding a *distributed switched network*. As a consequence, each network switch has one or more end node devices directly connected to it, thus forming a network node. These nodes are directly connected to other nodes without indirectly going through some external switch, giving rise to another popular name for these networks—*direct networks*.

The topology for distributed switched networks takes on a form much different from centralized switched networks in that end nodes are connected across the area of the switch fabric, not just at one or two of the peripheral edges of the fabric. This causes the number of switches in the system to be equal to the total number of nodes. A quite obvious way of interconnecting nodes consists of connecting a dedicated link between each node and every other node in the network. This *fully connected* topology provides the best connectivity (full connectivity in fact), but it is more costly than a crossbar network, as the following example shows.

Example Compute the cost of interconnecting N nodes using a fully connected topology relative to doing so using a crossbar topology. Consider separately the relative cost of the unidirectional links and the relative cost of the switches. Switch cost is assumed to grow quadratically with the number of unidirectional ports for $k \times k$ switches but to grow only linearly with $1 \times k$ switches.

Answer The crossbar topology requires an $N \times N$ switch, so the switch cost is proportional to N^2. The link cost is $2N$, which accounts for the unidirectional links from the end nodes to the centralized crossbar, and *vice versa*. In the fully connected topology, two sets of $1 \times (N-1)$ switches (possibly merged into one set) are used in each of the N nodes to connect nodes directly to and from all other nodes. Thus, the total switch cost for all N nodes is proportional to $2N(N-1)$. Regarding link cost, each of the N nodes requires two unidirectional links in opposite directions between its end node device and its local switch. In addition, each of the N nodes has $N-1$ unidirectional links from its local switch to other switches distributed across all the other end nodes. Thus, the total number of unidirectional links is $2N+N(N-1)$, which is equal to $N(N+1)$ for all N nodes. The relative costs of the fully connected topology with respect to the crossbar is, therefore, the following:

$$\text{Relative cost}_{switches} = 2N(N-1)/N^2 = 2(N-1)/N = 2(1-1/N)$$

$$\text{Relative cost}_{links} = N(N+1)/2N = (N+1)/2$$

As the number of interconnected devices increases, the switch cost of the fully connected topology is nearly double the crossbar, with both being very high (i.e., quadratic growth). Moreover, the fully connected topology always has higher relative link cost, which grows linearly with the number of nodes. Again, keep in mind that end node links are different from switch links in their length and packaging, particularly for direct networks, so they usually have different associated costs. Despite its higher cost, the fully connected topology provides no extra performance benefits over the crossbar as both are nonblocking. Thus, crossbar networks are usually used in practice instead of fully connected networks.

A lower-cost alternative to fully connecting all nodes in the network is to directly connect nodes in sequence along a *ring* topology, as shown in Figure F.13. For bidirectional rings, each of the N nodes now uses only 3×3 switches and just two bidirectional network links (shared by neighboring nodes), for a total of N switches and N bidirectional network links. This linear cost excludes the N injection-reception bidirectional links required within nodes.

Unlike shared-media networks, rings can allow many simultaneous transfers: the first node can send to the second while the second sends to the third, and so on. However, as dedicated links do not exist between logically nonadjacent node pairs, packets must hop across intermediate nodes before arriving at their destination, increasing their transport latency. For bidirectional rings, packets can be transported in either direction, with the shortest path to the destination usually being the one selected. In this case, packets must travel $N/4$ network switch hops, on average, with total switch hop count being one more to account for the local switch at the packet source node. Along the way, packets may block on network resources due to other packets contending for the same resources simultaneously.

Fully connected and ring-connected networks delimit the two extremes of distributed switched topologies, but there are many points of interest in between for a given set of cost-performance requirements. Generally speaking, the ideal switched-media topology has cost approaching that of a ring but performance

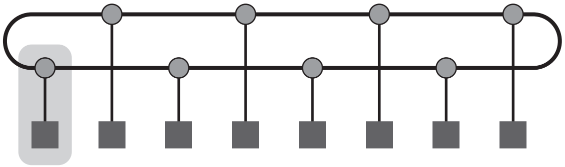
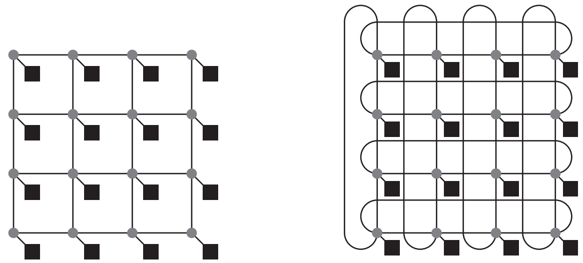
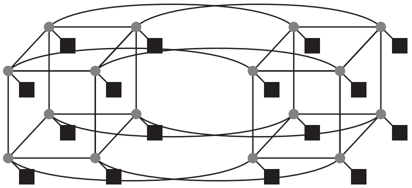


Figure F.13 A ring network topology, folded to reduce the length of the longest link. Shaded circles represent switches, and black squares represent end node devices. The gray rectangle signifies a network node consisting of a switch, a device, and its connecting link.



(A) 2D grid or mesh of 16 nodes

(B) 2D torus of 16 nodes



(C) Hypercube of 16 nodes ($16 = 2^4$ so $n = 4$)

Figure F.14 Direct network topologies that have appeared in commercial systems, mostly supercomputers.
The shaded circles represent switches, and the black squares represent end node devices. Switches have many bidirectional network links, but at least one link goes to the end node device. These basic topologies can be supplemented with extra links to improve performance and reliability. For example, connecting the switches on the periphery of the 2D mesh, shown in (a), using the unused ports on each switch forms a 2D torus, shown in (b). The hypercube topology, shown in (c) is an n-dimensional interconnect for 2^n nodes, requiring $n+1$ ports per switch: one for the n nearest neighbor nodes and one for the end node device.

approaching that of a fully connected topology. Figure F.14 illustrates three popular direct network topologies commonly used in systems spanning the cost-performance spectrum. All of them consist of sets of nodes arranged along multiple dimensions with a regular interconnection pattern among nodes that can be expressed mathematically. In the *mesh* or *grid* topology, all the nodes in each dimension form a linear array. In the *torus* topology, all the nodes in each dimension form a ring. Both of these topologies provide direct communication to neighboring nodes with the aim of reducing the number of hops suffered by packets in the network with respect to the ring. This is achieved by providing greater connectivity through additional dimensions, typically no more than three in commercial systems. The *hypercube* or *n-cube* topology is a particular case of the mesh in which only two nodes are interconnected along each dimension, leading to a number of dimensions, n, that must be large enough to interconnect all N nodes in the system (i.e., $n = \log_2 N$). The hypercube provides better connectivity than meshes and tori at the expense of higher link and switch costs, in terms of the number of links and number of ports per node.

Example Compute the cost of interconnecting N devices using a torus topology relative to doing so using a fat tree topology. Consider separately the relative cost of the bidirectional links and the relative cost of the switches—which is assumed to grow quadratically with the number of bidirectional ports. Provide an approximate expression for the case of switches being similar in size.

Answer Using $k \times k$ switches, the fat tree requires $2\,N/k\,(\log_{k/2} N)$ switches, assuming the last stage (the root) has the same number of switches as each of the other stages. Given that the number of bidirectional ports in each switch is k (i.e., there are k input ports and k output ports for a $k \times k$ switch) and that the switch cost grows quadratically with this, total network switch cost is proportional to $2kN \log_{k/2} N$. The link cost is $N \log_{k/2} N$ as each of the $\log_{k/2} N$ stages requires N bidirectional links, including those between the devices and the fat tree. The torus requires as many switches as nodes, each of them having $2n+1$ bidirectional ports, including the port to attach the communicating device, where n is the number of dimensions. Hence, total switch cost for the torus is $(2n+1)^2 N$. Each of the torus nodes requires $2n+1$ bidirectional links for the n different dimensions and the connection for its end node device, but as the dimensional links are shared by two nodes, the total number of links is $(2n/2+1)N = (n+1)N$ bidirectional links for all N nodes. Thus, the relative costs of the torus topology with respect to the fat tree are

$$\text{Relative cost}_{switches} = (2n+1)^2 N/2kN \log_{k/2} N = (2n+1)^2/2k \log_{k/2} N$$

$$\text{Relative cost}_{links} = (n+1)N/N \log_{k/2} N = (n+1)/\log_{k/2} N$$

When switch sizes are similar, $2n+1 \cong k$. In this case, the relative cost is

$$\text{Relative cost}_{switches} = (2n+1)^2/2k \log_{k/2} N = (2n+1)/2 \log_{k/2} N = k/2 \log_{k/2} N$$

When the number of switch ports (also called *switch degree*) is small, tori have lower cost, particularly when the number of dimensions is low. This is an especially useful property when N is large. On the other hand, when larger switches and/or a high number of tori dimensions are used, fat trees are less costly and preferable. For example, when interconnecting 256 nodes, a fat tree is four times more expensive in terms of switch and link costs when 4 × 4 switches are used. This higher cost is compensated for by lower network contention, on average. The fat tree is comparable in cost to the torus when 8 × 8 switches are used (e.g., for interconnecting 256 nodes). For larger switch sizes beyond this, the torus costs more than the fat tree as each node includes a switch. This cost can be amortized by connecting multiple end node devices per switch, called *bristling*.

The topologies depicted in Figure F.14 all have in common the interesting characteristic of having their network links arranged in several orthogonal dimensions in a regular way. In fact, these topologies all happen to be particular instances of a larger class of direct network topologies known as *k*-ary *n*-cubes, where *k* signifies the number of nodes interconnected in each of the *n* dimensions. The symmetry and regularity of these topologies simplify network implementation (i.e, packaging) and packet routing as the movement of a packet along a given network dimension does not modify the number of remaining hops in any other dimension toward its destination. As we will see in the next section, this topological property can be readily exploited by simple routing algorithms.

Like their indirect counterpart, direct networks can introduce blocking among packets that concurrently request the same path, or part of it. The only exception is fully connected networks. The same way that the number of stages and switch hops in indirect networks can be reduced by using larger switches, the hop count in direct networks can likewise be reduced by increasing the number of topological dimensions via increased switch degree.

It may seem to be a good idea always to maximize the number of dimensions for a system of a certain size and switch cost. However, this is not necessarily the case. Most electronic systems are built within our three-dimensional (3D) world using planar (2D) packaging technology such as integrated circuit chips, printed circuit boards, and backplanes. Direct networks with up to three dimensions can be implemented using relatively short links within this 3D space, independent of system size. Links in higher-dimensioned networks would require increasingly longer wires or fiber. This increase in link length with system size is also indicative of MINs, including fat trees, which require either long links within all the stages or increasingly longer links as more stages are added. As we saw in the first example given in Section F.2, flow-controlled buffers increase in size proportionally to link length, thus requiring greater silicon area. This is among the reasons why the supercomputer with the largest number of compute nodes existing in 2005, the IBM Blue Gene/L, implemented a 3D torus network for interprocessor communication. A fat tree would have required much longer links, rendering a 64K node system less feasible. This highlights the importance of correctly selecting the proper network topology that meets system requirements.

Besides link length, other constraints derived from implementing the topology may also limit the degree to which a topology can scale. These are available *pin-out* and achievable *bisection bandwidth*. Pin count is a local restriction on the bandwidth of a chip, printed circuit board, and backplane (or chassis) connector. In a direct network that integrates processor cores and switches on a single chip or multichip module, pin bandwidth is used both for interfacing with main memory and for implementing node links. In this case, limited pin count could reduce the number of switch ports or bit lines per link. In an indirect network, switches are implemented separately from processor cores, allowing most of the pins to be dedicated to communication bandwidth. However, as switches are grouped onto boards, the aggregate of all input-output links of the switch fabric on a board for a given topology must not exceed the board connector pin-outs.

The bisection bandwidth is a more global restriction that gives the interconnect density and bandwidth that can be achieved by a given implementation (packaging) technology. Interconnect density and clock frequency are related to each other: When wires are packed closer together, crosstalk and parasitic capacitance increase, which usually impose a lower clock frequency. For example, the availability and spacing of metal layers limit wire density and frequency of on-chip networks, and copper track density limits wire density and frequency on a printed circuit board. To be implementable, the topology of a network must not exceed the available bisection bandwidth of the implementation technology. Most networks implemented to date are constrained more so by pin-out limitations rather than bisection bandwidth, particularly with the recent move to blade-based systems. Nevertheless, bisection bandwidth largely affects performance.

For a given topology, bisection bandwidth, $BW_{Bisection}$, is calculated by dividing the network into two roughly equal parts—each with half the nodes—and summing the bandwidth of the links crossing the imaginary dividing line. For nonsymmetric topologies, bisection bandwidth is the smallest of all pairs of equal-sized divisions of the network. For a fully connected network, the bisection bandwidth is proportional to $N^2/2$ unidirectional links (or $N^2/4$ bidirectional links), where N is the number of nodes. For a bus, bisection bandwidth is the bandwidth of just the one shared half-duplex link. For other topologies, values lie in between these two extremes. Network injection and reception bisection bandwidth is commonly used as a reference value, which is $N/2$ for a network with N injection and reception links, respectively. Any network topology that provides this bisection bandwidth is said to have *full bisection bandwidth.*

Figure F.15 summarizes the number of switches and links required, the corresponding switch size, the maximum and average switch hop distances between nodes, and the bisection bandwidth in terms of links for several topologies discussed in this section for interconnecting 64 nodes.

Evaluation category	**Bus**	**Ring**	**2D mesh**	**2D torus**	**Hypercube**	**Fat tree**	**Fully connected**
Performance							
$BW_{Bisection}$ in # links	1	2	8	16	32	32	1024
Max (ave.) hop count	1 (1)	32 (16)	14 (7)	8 (4)	6 (3)	11 (9)	1 (1)
Cost							
I/O ports per switch	NA	3	5	5	7	4	64
Number of switches	NA	64	64	64	64	192	64
Number of net. links	1	64	112	128	192	320	2016
Total number of links	1	128	176	192	256	384	2080

Figure F.15 Performance and cost of several network topologies for 64 nodes. The bus is the standard reference at unit network link cost and bisection bandwidth. Values are given in terms of bidirectional links and ports. Hop count includes a switch and its output link, but not the injection link at end nodes. Except for the bus, values are given for the number of network links and total number of links, including injection/reception links between end node devices and the network.

Effects of Topology on Network Performance

Switched network topologies require packets to take one or more hops to reach their destination, where each hop represents the transport of a packet through a switch and one of its corresponding links. Interestingly, each switch and its corresponding links can be modeled as a black box network connecting more than two devices, as was described in the previous section, where the term "devices" here refers to end nodes or other switches. The only differences are that the sending and receiving overheads are null through the switches, and the routing, switching, and arbitration delays are not cumulative but, instead, are delays associated with each switch.

As a consequence of the above, if the average packet has to traverse d hops to its destination, then $T_R + T_A + T_S = (T_r + T_a + T_s) \times d$, where T_r, T_a, and T_s are the routing, arbitration, and switching delays, respectively, of a switch. With the assumption that pipelining over the network is staged on each hop at the packet level (this assumption will be challenged in the next section), the transmission delay is also increased by a factor of the number of hops. Finally, with the simplifying assumption that all injection links to the first switch or stage of switches and all links (including reception links) from the switches have approximately the same length and delay, the total propagation delay through the network $T_{\text{TotalProp}}$ is the propagation delay through a single link, T_{LinkProp}, multiplied by $d + 1$, which is the hop count plus one to account for the injection link. Thus, the best-case lower-bound expression for average packet latency in the network (i.e., the latency in the absence of contention) is given by the following expression:

$$\text{Latency} = \text{Sending overhead} + T_{\text{LinkProp}} \times (d+1) + (T_r + T_a + T_s) \times d + \frac{\text{Packet size}}{\text{Bandwidth}} \times (d+1) + \text{Receiving overhead}$$

Again, the expression on page F-40 assumes that switches are able to pipeline packet transmission at the packet level.

Following the method presented previously, we can estimate the best-case upper bound for effective bandwidth by finding the narrowest section of the end-to-end network pipe. Focusing on the internal network portion of that pipe, network bandwidth is determined by the blocking properties of the topology. Non-blocking behavior can be achieved only by providing many alternative paths between every source-destination pair, leading to an aggregate network bandwidth that is many times higher than the aggregate network injection or reception bandwidth. This is quite costly. As this solution usually is prohibitively expensive, most networks have different degrees of blocking, which reduces the utilization of the aggregate bandwidth provided by the topology. This, too, is costly but not in terms of performance.

The amount of blocking in a network depends on its topology and the traffic distribution. Assuming the bisection bandwidth, $\text{BW}_{\text{Bisection}}$, of a topology is implementable (as typically is the case), it can be used as a constant measure of the maximum degree of blocking in a network. In the ideal case, the network always achieves full bisection bandwidth irrespective of the traffic behavior, thus transferring the bottlenecking point to the injection or reception links. However, as packets destined to locations in the other half of the network necessarily must cross the bisection links, those links pose as potential bottleneck links—potentially reducing the network bandwidth to below full bisection bandwidth. Fortunately, not all of the traffic must cross the network bisection, allowing more of the aggregate network bandwidth provided by the topology to be utilized. Also, network topologies with a higher number of bisection links tend to have less blocking as more alternative paths are possible to reach destinations and, hence, a higher percentage of the aggregate network bandwidth can be utilized. If only a fraction of the traffic must cross the network bisection, as captured by a *bisection traffic fraction* parameter γ ($0 < \gamma \leq 1$), the network pipe at the bisection is, effectively, widened by the reciprocal of that fraction, assuming a traffic distribution that loads the bisection links at least as heavily, on average, as other network links. This defines the upper limit on achievable network bandwidth, $\text{BW}_{\text{Network}}$:

$$\text{BW}_{\text{Network}} = \frac{\text{BW}_{\text{Bisection}}}{\gamma}$$

Accordingly, the expression for effective bandwidth becomes the following when network topology is taken into consideration:

$$\text{Effective bandwidth} = \min\left(N \times \text{BW}_{\text{LinkInjection}}, \frac{\text{BW}_{\text{Bisection}}}{\gamma}, \sigma \times N \times \text{BW}_{\text{LinkReception}}\right)$$

It is important to note that γ depends heavily on the traffic patterns generated by applications. It is a measured quantity or calculated from detailed traffic analysis.

Example A common communication pattern in scientific programs is to have nearest neighbor elements of a two-dimensional array to communicate in a given direction. This pattern is sometimes called *NEWS communication*, standing for north, east, west, and south—the directions on a compass. Map an 8×8 array of elements one-to-one onto 64 end node devices interconnected in the following topologies: bus, ring, 2D mesh, 2D torus, hypercube, fully connected, and fat tree. How long does it take in the best case for each node to send one message to its northern neighbor and one to its eastern neighbor, assuming packets are allowed to use any minimal path provided by the topology? What is the corresponding effective bandwidth? Ignore elements that have no northern or eastern neighbors. To simplify the analysis, assume that all networks experience unit packet transport time for each network hop—that is, T_{LinkProp}, T_r, T_a, T_s, and packet transmission time for each hop sum to one. Also assume the delay through injection links is included in this unit time, and sending/receiving overhead is null.

Answer This communication pattern requires us to send $2 \times (64 - 8)$ or 112 total packets—that is, 56 packets in each of the two communication phases: northward and eastward. The number of hops suffered by packets depends on the topology. Communication between sources and destinations are one-to-one, so σ is 100%.

The injection and reception bandwidth cap the effective bandwidth to a maximum of 64 BW units (even though the communication pattern requires only 56 BW units). However, this maximum may get scaled down by the achievable network bandwidth, which is determined by the bisection bandwidth and the fraction of traffic crossing it, γ, both of which are topology dependent. Here are the various cases:

- *Bus*—The mapping of the 8 × 8 array elements to nodes makes no difference for the bus as all nodes are equally distant at one hop away. However, the 112 transfers are done sequentially, taking a total of 112 time units. The bisection bandwidth is 1, and γ is 100%. Thus, effective bandwidth is only 1 BW unit.
- *Ring*—Assume the first row of the array is mapped to nodes 0 to 7, the second row to nodes 8 to 15, and so on. It takes just one time unit for all nodes simultaneously to send to their eastern neighbor (i.e., a transfer from node i to node $i+1$). With this mapping, the northern neighbor for each node is exactly eight hops away so it takes eight time units, which also is done in parallel for all nodes. Total communication time is, therefore, 9 time units. The bisection bandwidth is 2 bidirectional links (assuming a bidirectional ring), which is less than the full bisection bandwidth of 32 bidirectional links. For eastward communication, because only 2 of the eastward 56 packets must cross the bisection in the worst case, the bisection links do not pose as bottlenecks. For northward communication, 8 of the 56 packets must cross the two bisection links, yielding a γ of $10/112 = 8.93\%$. Thus, the network bandwidth is $2/.0893 = 22.4$ BW units. This limits the effective bandwidth at 22.4 BW units as well, which is less than half the bandwidth required by the communication pattern.
- *2D mesh*—There are eight rows and eight columns in our grid of 64 nodes, which is a perfect match to the NEWS communication. It takes a total of just 2 time units for all nodes to send simultaneously to their northern neighbors followed by simultaneous communication to their eastern neighbors. The bisection bandwidth is 8 bidirectional links, which is less than full bisection bandwidth. However, the perfect matching of this nearest neighbor communication pattern on this topology allows the maximum effective bandwidth to be achieved regardless. For eastward communication, 8 of the 56 packets must cross the bisection in the worst case, which does not exceed the bisection bandwidth. None of the northward communications crosses the same network bisection, yielding a γ of $8/112 = 7.14\%$ and a network bandwidth of $8/0.0714 = 112$ BW units. The effective bandwidth is, therefore, limited by the communication pattern at 56 BW units as opposed to the mesh network.
- *2D torus*—Wrap-around links of the torus are not used for this communication pattern, so the torus has the same mapping and performance as the mesh.

- *Hypercube*—Assume elements in each row are mapped to the same location within the eight 3-cubes comprising the hypercube such that consecutive row elements are mapped to nodes only one hop away. Northern neighbors can be similarly mapped to nodes only one hop away in an orthogonal dimension. Thus, the communication pattern takes just 2 time units. The hypercube provides full bisection bandwidth of 32 links, but at most only 8 of the 112 packets must cross the bisection. Thus, effective bandwidth is limited only by the communication pattern to be 56 BW units, not by the hypercube network.
- *Fully connected*—Here, nodes are equally distant at one hop away, regardless of the mapping. Parallel transfer of packets in both the northern and eastern directions would take only 1 time unit if the injection and reception links could source and sink two packets at a time. As this is not the case, 2 time units are required. Effective bandwidth is limited by the communication pattern at 56 BW units, so the 1024 network bisection links largely go underutilized.
- *Fat tree*—Assume the same mapping of elements to nodes as is done for the ring and the use of switches with eight bidirectional ports. This allows simultaneous communication to eastern neighbors that takes at most three hops and, therefore, 3 time units through the three bidirectional stages interconnecting the eight nodes in each of the eight groups of nodes. The northern neighbor for each node resides in the adjacent group of eight nodes, which requires five hops, or 5 time units. Thus, the total time required on the fat tree is 8 time units. The fat tree provides full bisection bandwidth, so in the worst case of half the traffic needing to cross the bisection, an effective bandwidth of 56 BW units (as limited by the communication pattern and not by the fattree network) is achieved when packets are continually injected.

The above example should not lead one to the wrong conclusion that meshes are just as good as tori, hypercubes, fat trees, and other networks with higher bisection bandwidth. A number of simplifications that benefit low-bisection networks were assumed to ease the analysis. In practice, packets typically are larger than the link width and occupy links for many more than just one network cycle. Also, many communication patterns do not map so cleanly to the 2D mesh network topology; instead, usually they are more global and irregular in nature. These and other factors combine to increase the chances of packets blocking in low-bisection networks, increasing latency and reducing effective bandwidth.

To put this discussion on topologies into further perspective, Figure F.16 lists various attributes of topologies used in commercial high-performance computers.

F.5 Network Routing, Arbitration, and Switching

Routing, arbitration, and switching are performed at every switch along a packet's path in a switched media network, no matter what the network topology. Numerous interesting techniques for accomplishing these network functions have been

Company	System [network] name	Max. number of nodes [× # CPUs]	Basic network topology	Injection [reception] node BW in MB/sec	# of data bits per link per direction	Raw network link BW per direction in MB/sec	Raw network bisection BW (bidirectional) in GB/sec
Intel	ASCI Red Paragon	4816 [×2]	2D mesh 64 × 64	400 [400]	16 bits	400	51.2
IBM	ASCI White SP Power3 [Colony]	512 [×16]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	500 [500]	8 bits (+1 bit of control)	500	256
Intel	Thunder Itanium2 Tiger4 [QsNet$^{\text{II}}$]	1024 [×4]	Fat tree with 8-port bidirectional switches	928 [928]	8 bits (+2 of control for 4b/5b encoding)	1333	1365
Cray	XT3 [SeaStar]	30,508 [×1]	3D torus 40 × 32 × 24	3200 [3200]	12 bits	3800	5836.8
Cray	X1E	1024 [×1]	4-way bristled 2D torus (~23 × 11) with express links	1600 [1600]	16 bits	1600	51.2
IBM	ASC Purple pSeries 575 [Federation]	>1280 [×8]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	2000 [2000]	8 bits (+2 bits of control for novel 5b/6b encoding scheme)	2000	2560
IBM	Blue Gene/L eServer Sol. [Torus Net.]	65,536 [×2]	3D torus 32 × 32 × 64	612.5 [1050]	1 bit (bit serial)	175	358.4

Figure F.16 Topological characteristics of interconnection networks used in commercial high-performance machines.

proposed in the literature. In this section, we focus on describing a representative set of approaches used in commercial systems for the more commonly used network topologies. Their impact on performance is also highlighted.

Routing

The *routing algorithm* defines which network path, or paths, are allowed for each packet. Ideally, the routing algorithm supplies shortest paths to all packets such that traffic load is evenly distributed across network links to minimize contention. However, some paths provided by the network topology may not be allowed in order to guarantee that all packets can be delivered, no matter what the traffic behavior. Paths that have an unbounded number of allowed nonminimal hops from packet sources, for instance, may result in packets never reaching their destinations. This situation is referred to as *livelock*. Likewise, paths that cause a set of packets to block in the network forever waiting only for network resources (i.e., links or associated buffers) held by other packets in the set also prevent packets from reaching their destinations. This situation is referred to as *deadlock*. As deadlock arises due to the finiteness of network resources, the probability of its occurrence increases with increased network traffic and decreased availability of network resources. For the network to function properly, the routing algorithm must guard against this anomaly, which can occur in various forms—for example, routing deadlock, request-reply (protocol) deadlock, and fault-induced (reconfiguration) deadlock, etc. At the same time, for the network to provide the highest possible performance, the routing algorithm must be efficient—allowing as many routing options to packets as there are paths provided by the topology, in the best case.

The simplest way of guarding against livelock is to restrict routing such that only minimal paths from sources to destinations are allowed or, less restrictively, only a limited number of nonminimal hops. The strictest form has the added benefit of consuming the minimal amount of network bandwidth, but it prevents packets from being able to use alternative nonminimal paths in case of contention or faults along the shortest (minimal) paths.

Deadlock is more difficult to guard against. Two common strategies are used in practice: avoidance and recovery. In *deadlock avoidance*, the routing algorithm restricts the paths allowed by packets to only those that keep the global network state deadlock-free. A common way of doing this consists of establishing an ordering between a set of resources—the minimal set necessary to support network full access—and granting those resources to packets in some total or partial order such that cyclic dependency cannot form on those resources. This allows an escape path always to be supplied to packets no matter where they are in the network to avoid entering a deadlock state. In *deadlock recovery*, resources are granted to packets without regard for avoiding deadlock. Instead, as deadlock is possible, some mechanism is used to detect the likely existence of deadlock. If detected, one or more packets are removed from resources in the deadlock set—possibly by regressively dropping the packets or by progressively redirecting the packets onto special deadlock recovery resources. The freed network resources are then granted to other packets needing them to resolve the deadlock.

Let us consider routing algorithms designed for distributed switched networks. Figure F.17(a) illustrates one of many possible deadlocked configurations for packets within a region of a 2D mesh network. The routing algorithm can avoid all such deadlocks (and livelocks) by allowing only the use of minimal paths that cross the network dimensions in some total order. That is, links of a given dimension are not supplied to a packet by the routing algorithm until no other links are needed by the packet in all of the preceding dimensions for it to reach its

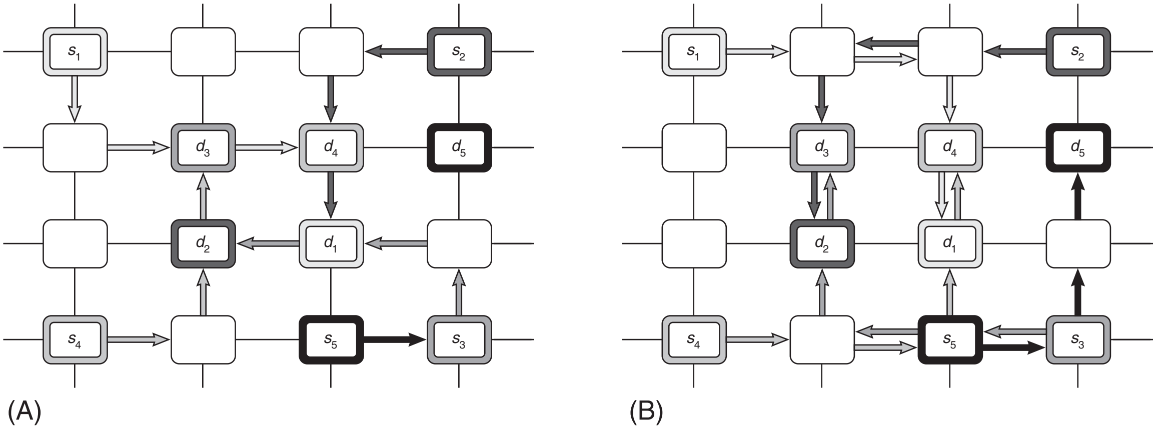


Figure F.17 A mesh network with packets routing from sources, s_i, to destinations, d_i. (a) Deadlock forms from packets destined to d_1 through d_4 blocking on others in the same set that fully occupy their requested buffer resources one hop away from their destinations. This deadlock cycle causes other packets needing those resources also to block, like packets from s_5 destined to d_5 that have reached node s_3. (b) Deadlock is avoided using dimension-order routing. In this case, packets exhaust their routes in the X dimension before turning into the Y dimension in order to complete their routing.

destination. This is illustrated in Figure F.17(b), where dimensions are crossed in *XY* dimension order. All the packets must follow the same order when traversing dimensions, exiting a dimension only when links are no longer required in that dimension. This well-known algorithm is referred to as *dimension-order routing* (DOR) or *e-cube routing* in hypercubes. It is used in many commercial systems built from distributed switched networks and on-chip networks. As this routing algorithm always supplies the same path for a given source-destination pair, it is a *deterministic routing* algorithm.

Crossing dimensions in order on some minimal set of resources required to support network full access avoids deadlock in meshes and hypercubes. However, for distributed switched topologies that have wrap-around links (e.g., rings and tori), a total ordering on a minimal set of resources within each dimension is also needed if resources are to be used to full capacity. Alternatively, some empty resources or *bubbles* along the dimensions would be required to remain below full capacity and avoid deadlock. To allow full access, either the physical links must be duplicated or the logical buffers associated with each link must be duplicated, resulting in *physical channels* or *virtual channels*, respectively, on which the ordering is done. Ordering is not necessary on all network resources to avoid deadlock—it is needed only on some minimal set required to support network full access (i.e., some *escape resource set*). Routing algorithms based on this technique (called Duato's protocol) can be defined that allow alternative paths provided by the topology to be used for a given source-destination pair in addition to the escape resource set. One of those allowed paths must be selected, preferably the most efficient one. Adapting the path in response to prevailing network traffic conditions enables the aggregate network bandwidth to be better utilized and contention to be reduced. Such routing capability is referred to as *adaptive routing* and is used in many commercial systems.

Example How many of the possible dimensional turns are eliminated by dimension-order routing on an n-dimensional mesh network? What is the fewest number of turns that actually need to be eliminated while still maintaining connectedness and deadlock freedom? Explain using a 2D mesh network.

Answer The dimension-order routing algorithm eliminates exactly half of the possible dimensional turns as it is easily proven that all turns from any lower-ordered dimension into any higher-ordered dimension are allowed, but the converse is not true. For example, of the eight possible turns in the 2D mesh shown in Figure F.17, the four turns from $X+$ to $Y+$, $X+$ to $Y-$, $X-$ to $Y+$, and $X-$ to $Y-$ are allowed, where the signs (+ or −) refer to the direction of travel within a dimension. The four turns from $Y+$ to $X+$, $Y+$ to $X-$, $Y-$ to $X+$, and $Y-$ to $X-$ are disallowed turns. The elimination of these turns prevents cycles of any kind from forming—and, thus, avoids deadlock—while keeping the network connected. However, it does so at the expense of not allowing any routing adaptivity.

The *Turn Model* routing algorithm proves that the minimum number of eliminated turns to prevent cycles and maintain connectedness is a quarter of the possible turns, but the right set of turns must be chosen. Only some particular set of eliminated turns allow both requirements to be satisfied. With the elimination of the wrong set of a quarter of the turns, it is possible for combinations of allowed turns to emulate the eliminated ones (and, thus, form cycles and deadlock) or for the network not to be connected. For the 2D mesh, for example, it is possible to eliminate only the two turns ending in the westward direction (i.e., $Y+$ to $X-$ and $Y-$ to $X-$) by requiring packets to start their routes in the westward direction (if needed) to maintain connectedness. Alternatives to this west-first routing for 2D meshes are negative-first routing and north-last routing. For these, the extra quarter of turns beyond that supplied by DOR allows for partial adaptivity in routing, making these adaptive routing algorithms.

Routing algorithms for centralized switched networks can similarly be defined to avoid deadlocks by restricting the use of resources in some total or partial order. For fat trees, resources can be totally ordered along paths starting from the input leaf stage upward to the root and then back down to the output leaf stage. The routing algorithm can allow packets to use resources in increasing partial order, first traversing up the tree until they reach some *least common ancestor* (LCA) of the source and destination, and then back down the tree until they reach their destinations. As there are many least common ancestors for a given destination, multiple alternative paths are allowed while going up the tree, making the routing algorithm adaptive. However, only a single deterministic path to the destination is provided by the fat tree topology from a least common ancestor. This *self-routing* property is common to many MINs and can be readily exploited: The switch output port at each stage is given simply by shifts of the destination node address.

More generally, a tree graph can be mapped onto any topology—whether direct or indirect—and links between nodes at the same tree level can be allowed by assigning directions to them, where "up" designates paths moving toward the tree root and "down" designates paths moving away from the root node. This allows for generic *up*/down* routing* to be defined on any topology such that packets follow paths (possibly adaptively) consisting of zero or more up links followed by zero or more down links to their destination. Up/down ordering prevents cycles from forming, avoiding deadlock. This routing technique was used in Autonet—a self-configuring switched LAN—and in early Myrinet SANs.

Routing algorithms are implemented in practice by a combination of the routing information placed in the packet header by the source node and the routing control mechanism incorporated in the switches. For *source routing*, the entire routing path is precomputed by the source—possibly by table lookup—and placed in the packet header. This usually consists of the output port or ports supplied for each switch along the predetermined path from the source to the destination, which can be stripped off by the routing control mechanism at each switch. An additional bit field can be included in the header to signify whether adaptive routing is allowed (i.e., that any one of the supplied output ports can be used). For *distributed routing*, the routing information usually consists of the destination address. This is used by the routing control mechanism in each switch along the path to determine the next output port, either by computing it using a finite-state machine or by looking it up in a local routing table (i.e., forwarding table). Compared to distributed routing, source routing simplifies the routing control mechanism within the network switches, but it requires more routing bits in the header of each packet, thus increasing the header overhead.

Arbitration

The *arbitration algorithm* determines when requested network paths are available for packets. Ideally, arbiters maximize the matching of free network resources and packets requesting those resources. At the switch level, arbiters maximize the matching of free output ports and packets located in switch input ports requesting those output ports. When all requests cannot be granted simultaneously, switch arbiters resolve conflicts by granting output ports to packets in a fair way such that *starvation* of requested resources by packets is prevented. This could happen to packets in shorter queues if a serve-longest-queue (SLQ) scheme is used. For packets having the same priority level, simple round-robin (RR) or age-based schemes are sufficiently fair and straightforward to implement.

Arbitration can be distributed to avoid centralized bottlenecks. A straightforward technique consists of two phases: a request phase and a grant phase. Let us assume that each switch input port has an associated queue to hold incoming

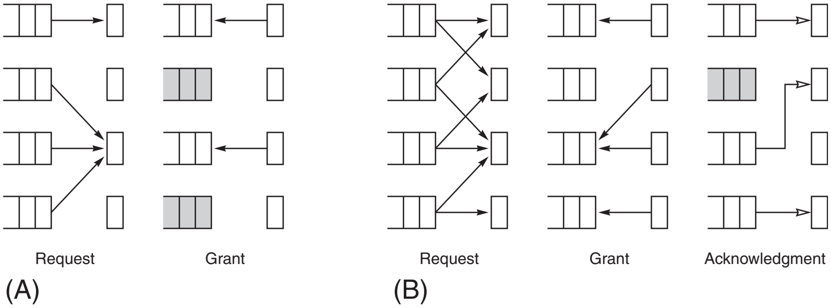


Figure F.18 Two arbitration techniques. (a) Two-phased arbitration in which two of the four input ports are granted requested output ports. (b) Three-phased arbitration in which three of the four input ports are successful in gaining the requested output ports, resulting in higher switch utilization.

packets and that each switch output port has an associated local arbiter implementing a round-robin strategy. Figure F.18(a) shows a possible set of requests for a four-port switch. In the *request phase*, packets at the head of each input port queue send a single request to the arbiters corresponding to the output ports requested by them. Then, each output port arbiter independently arbitrates among the requests it receives, selecting only one. In the *grant phase*, one of the requests to each arbiter is granted the requested output port. When two packets from different input ports request the same output port, only one receives a grant, as shown in the figure. As a consequence, some output port bandwidth remains unused even though all input queues have packets to transmit.

The simple two-phase technique can be improved by allowing several simultaneous requests to be made by each input port, possibly coming from different virtual channels or from multiple adaptive routing options. These requests are sent to different output port arbiters. By submitting more than one request per input port, the probability of matching increases. Now, arbitration requires three phases: request, grant, and acknowledgment. Figure F.18(b) shows the case in which up to two requests can be made by packets at each input port. In the request phase, requests are submitted to output port arbiters, and these arbiters select one of the received requests, as is done for the two-phase arbiter. Likewise, in the grant phase, the selected requests are granted to the corresponding requesters. Taking into account that an input port can submit more than one request, it may receive more than one grant. Thus, it selects among possibly multiple grants using some arbitration strategy such as round-robin. The selected grants are confirmed to the corresponding output port arbiters in the acknowledgment phase.

As can be seen in Figure F.18(b), it could happen that an input port that submits several requests does not receive any grants, while some of the requested ports remain free. Because of this, a second arbitration iteration can improve the probability of matching. In this iteration, only the requests corresponding to non-matched input and output ports are submitted. Iterative arbiters with multiple requests per input port are able to increase the utilization of switch output ports and, thus, the network link bandwidth. However, this comes at the expense of additional arbiter complexity and increased arbitration delay, which could increase the router clock cycle time if it is on the critical path.

Switching

The *switching technique* defines how connections are established in the network. Ideally, connections between network resources are established or "switched in" only for as long as they are actually needed and exactly at the point that they are ready and needed to be used, considering both time and space. This allows efficient use of available network bandwidth by competing traffic flows and minimal latency. Connections at each hop along the topological path allowed by the routing algorithm and granted by the arbitration algorithm can be established in three basic ways: prior to packet arrival using *circuit switching*, upon receipt of the entire packet using *store-and-forward packet switching*, or upon receipt of only portions of the packet with unit size no smaller than that of the packet header using *cut-through packet switching*.

Circuit switching establishes a circuit *a priori* such that network bandwidth is allocated for packet transmissions along an entire source-destination path. It is possible to pipeline packet transmission across the circuit using staging at each hop along the path, a technique known as *pipelined circuit switching*. As routing, arbitration, and switching are performed only once for one or more packets, routing bits are not needed in the header of packets, thus reducing latency and overhead. This can be very efficient when information is continuously transmitted between devices for the same circuit setup. However, as network bandwidth is removed from the shared pool and preallocated regardless of whether sources are in need of consuming it or not, circuit switching can be very inefficient and highly wasteful of bandwidth.

Packet switching enables network bandwidth to be shared and used more efficiently when packets are transmitted intermittently, which is the more common case. Packet switching comes in two main varieties—store-and-forward and cutthrough switching, both of which allow network link bandwidth to be multiplexed on packet-sized or smaller units of information. This better enables bandwidth sharing by packets originating from different sources. The finer granularity of sharing, however, increases the overhead needed to perform switching: Routing, arbitration, and switching must be performed for every packet, and routing and flow control bits are required for every packet if flow control is used.

Store-and-forward packet switching establishes connections such that a packet is forwarded to the next hop in sequence along its source-destination path only after the entire packet is first stored (staged) at the receiving switch. As packets are completely stored at every switch before being transmitted, links are completely decoupled, allowing full link bandwidth utilization even if links have very different bandwidths. This property is very important in WANs, but the price to pay is packet latency; the total routing, arbitration, and switching delay is multiplicative with the number of hops, as we have seen in Section F.4 when analyzing performance under this assumption.

Cut-through packet switching establishes connections such that a packet can "cut through" switches in a pipelined manner once the header portion of the packet (or equivalent amount of payload trailing the header) is staged at receiving switches. That is, the rest of the packet need not arrive before switching in the granted resources. This allows routing, arbitration, and switching delay to be additive with the number of hops rather than multiplicative to reduce total packet latency. Cut-through comes in two varieties, the main differences being the size of the unit of information on which flow control is applied and, consequently, the buffer requirements at switches. *Virtual cut-through switching* implements flow control at the packet level, whereas *wormhole switching* implements it on flow units, or *flits*, which are smaller than the maximum packet size but usually at least as large as the packet header. Since wormhole switches need to be capable of storing only a small portion of a packet, packets that block in the network may span several switches. This can cause other packets to block on the links they occupy, leading to premature network saturation and reduced effective bandwidth unless some centralized buffer is used within the switch to store them—a technique called *buffered wormhole switching*. As chips can implement relatively large buffers in current technology, virtual cut-through is the more commonly used switching technique. However, wormhole switching may still be preferred in OCNs designed to minimize silicon resources.

Premature network saturation caused by wormhole switching can be mitigated by allowing several packets to share the physical bandwidth of a link simultaneously via time-multiplexed switching at the flit level. This requires physical links to have a set of virtual channels (i.e., the logical buffers mentioned previously) at each end, into which packets are switched. Before, we saw how virtual channels can be used to decouple physical link bandwidth from buffered packets in such a way as to avoid deadlock. Now, virtual channels are multiplexed in such a way that bandwidth is switched in and used by flits of a packet to advance even though the packet may share some links in common with a blocked packet ahead. This, again, allows network bandwidth to be used more efficiently, which, in turn, reduces the average packet latency.

Impact on Network Performance

Routing, arbitration, and switching can impact the packet latency of a loaded network by reducing the contention delay experienced by packets. For an unloaded network that has no contention, the algorithms used to perform routing and arbitration have no impact on latency other than to determine the amount of delay incurred in implementing those functions at switches—typically, the pin-to-pin latency of a switch chip is several tens of nanoseconds. The only change to the best-case packet latency expression given in the previous section comes from the switching technique. Store-and-forward packet switching was assumed before in which transmission delay for the entire packet is incurred on all d hops plus at the source node. For cut-through packet switching, transmission delay is pipelined across the network links comprising the packet's path at the granularity of the packet header instead of the entire packet. Thus, this delay component is reduced, as shown in the following lower-bound expression for packet latency:

$$\text{Latency} = \text{Sending overhead} + T_{\text{LinkProp}} \times (d+1) + (T_r + \tau_a + T_S) \times d + \frac{(\text{Packet} + (d \times \text{Header}))}{\text{Bandwidth}} + \text{Receiving overhead}$$

The effective bandwidth is impacted by how efficiently routing, arbitration, and switching allow network bandwidth to be used. The routing algorithm can distribute traffic more evenly across a loaded network to increase the utilization of the aggregate bandwidth provided by the topology—particularly, by the bisection links. The arbitration algorithm can maximize the number of switch output ports that accept packets, which also increases the utilization of network bandwidth. The switching technique can increase the degree of resource sharing by packets, which further increases bandwidth utilization. These combine to affect network bandwidth, $\text{BW}_{\text{Network}}$, by an *efficiency factor*, ρ, where $0 < \rho \leq 1$:

$$\text{BW}_{\text{Network}} = \rho \times \frac{\text{BW}_{\text{Bisection}}}{\gamma}$$

The efficiency factor, ρ, is difficult to calculate or to quantify by means other than simulation. Nevertheless, with this parameter we can estimate the best-case upper-bound effective bandwidth by using the following expression that takes into account the effects of routing, arbitration, and switching:

$$\text{Effective bandwidth} = \min\left(N \times \text{BW}_{\text{LinkInjection}},\ \rho \times \frac{\text{BW}_{\text{Bisection}}}{\gamma}, \sigma \times N \times \text{BW}_{\text{LinkReception}}\right)$$

We note that ρ also depends on how well the network handles the traffic generated by applications. For instance, ρ could be higher for circuit switching than for cut-through switching if large streams of packets are continually transmitted between a source-destination pair, whereas the converse could be true if packets are transmitted intermittently.

Example Compare the performance of deterministic routing versus adaptive routing for a 3D torus network interconnecting 4096 nodes. Do so by plotting latency versus applied load and throughput versus applied load. Also compare the efficiency of the best and worst of these networks. Assume that virtual cut-through switching, three-phase arbitration, and virtual channels are implemented. Consider separately the cases for two and four virtual channels, respectively. Assume that one of the virtual channels uses bubble flow control in dimension order so as to avoid deadlock; the other virtual channels are used either in dimension order (for deterministic routing) or minimally along shortest paths (for adaptive routing), as is done in the IBM Blue Gene/L torus network.

Answer It is very difficult to compute analytically the performance of routing algorithms given that their behavior depends on several network design parameters with complex interdependences among them. As a consequence, designers typically resort to cycle-accurate simulators to evaluate performance. One way to evaluate the effect of a certain design decision is to run sets of simulations over a range of network loads, each time modifying one of the design parameters of interest while keeping the remaining ones fixed. The use of synthetic traffic loads is quite frequent in these evaluations as it allows the network to stabilize at a certain working point and for behavior to be analyzed in detail. This is the method we use here (alternatively, trace-driven or execution-driven simulation can be used).

Figure F.19 shows the typical interconnection network performance plots. On the left, average packet latency (expressed in network cycles) is plotted as a function of applied load (traffic generation rate) for the two routing algorithms with two and four virtual channels each; on the right, throughput (traffic delivery rate) is similarly plotted. Applied load is normalized by dividing it by the number of nodes in the network (i.e., bytes per cycle per node). Simulations are run under the assumption of uniformly distributed traffic consisting of 256-byte packets, where flits are byte sized. Routing, arbitration, and switching delays are assumed to sum to 1 network cycle per hop while the time-of-flight delay over each link is assumed to be 10 cycles. Link bandwidth is 1 byte per cycle, thus providing results that are independent of network clock frequency.

As can be seen, the plots within each graph have similar characteristic shapes, but they have different values. For the latency graph, all start at the no-load latency

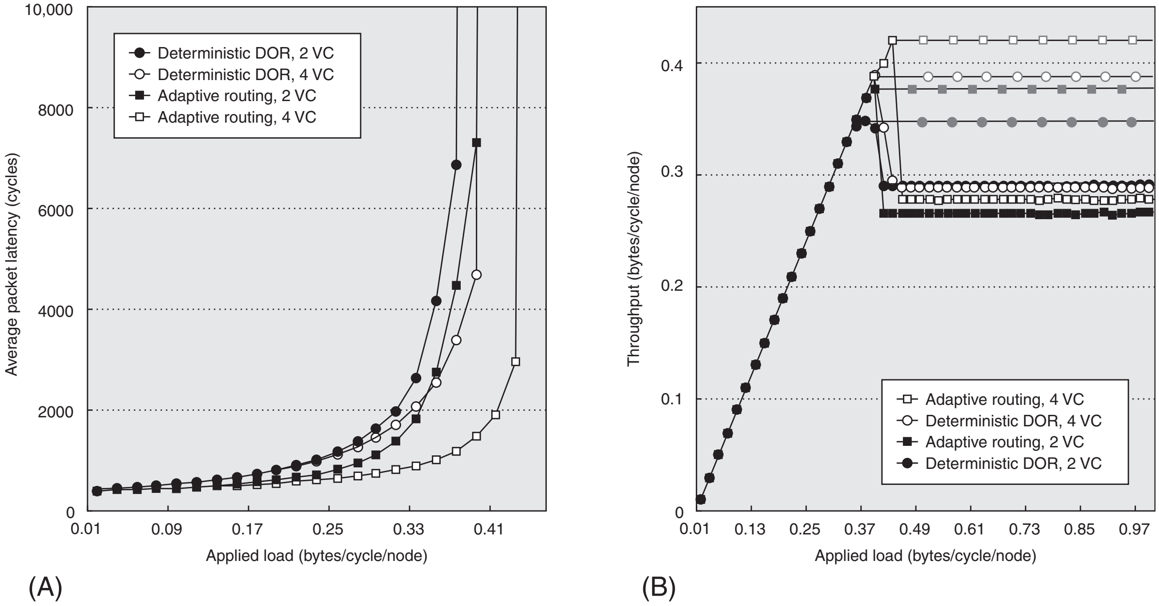


Figure F.19 Deterministic routing is compared against adaptive routing, both with either two or four virtual channels, assuming uniformly distributed traffic on a 4 K node 3D torus network with virtual cut-through switching and bubble flow control to avoid deadlock. (a) Average latency is plotted versus applied load, and (b) throughput is plotted versus applied load (the upper grayish plots show peak throughput, and the lower black plots show sustained throughput). Simulation data were collected by P. Gilabert and J. Flich at the Universidad Politècnica de València, Spain (2006).

as predicted by the latency expression given above, then slightly increase with traffic load as contention for network resources increases. At higher applied loads, latency increases exponentially, and the network approaches its saturation point as it is unable to absorb the applied load, = causing packets to queue up at their source nodes awaiting injection. In these simulations, the queues keep growing over time, making latency tend toward infinity. However, in practice, queues reach their capacity and trigger the application to stall further packet generation, or the application throttles itself waiting for acknowledgments/responses to outstanding packets. Nevertheless, latency grows at a slower rate for adaptive routing as alternative paths are provided to packets along congested resources.

For this same reason, adaptive routing allows the network to reach a higher peak throughput for the same number of virtual channels as compared to deterministic routing. At nonsaturation loads, throughput increases fairly linearly with applied load. When the network reaches its saturation point, however, it is unable to deliver traffic at the same rate at which traffic is generated. The saturation point, therefore, indicates the maximum achievable or "peak" throughput, which would be no more than that predicted by the effective bandwidth expression given above. Beyond saturation, throughput tends to drop as a consequence of massive head-of-line blocking across the network (as will be explained further in Section F.6), very much like cars tend to advance more slowly at rush hour. This is an important region of the throughput graph as it shows how significant of a performance drop the routing algorithm can cause if congestion management techniques (discussed briefly in Section F.7) are not used effectively. In this case, adaptive routing has more of a performance drop after saturation than deterministic routing, as measured by the postsaturation sustained throughput.

For both routing algorithms, more virtual channels (i.e., four) give packets a greater ability to pass over blocked packets ahead, allowing for a higher peak throughput as compared to fewer virtual channels (i.e., two). For adaptive routing with four virtual channels, the peak throughput of 0.43 bytes/cycle/node is near the maximum of 0.5 bytes/cycle/node that can be obtained with 100% efficiency (i.e., $\rho = 100\%$), assuming there is enough injection and reception bandwidth to make the network bisection the bottlenecking point. In that case, the network bandwidth is simply 100% times the network bisection bandwidth ($BW_{\text{Bisection}}$) divided by the fraction of traffic crossing the bisection (γ), as given by the expression above. Taking into account that the bisection splits the torus into two equally sized halves, γ is equal to 0.5 for uniform traffic as only half the injected traffic is destined to a node at the other side of the bisection. The $BW_{\text{Bisection}}$ for a 4096-node 3D torus network is $16 \times 16 \times 4$ unidirectional links times the link bandwidth (i.e., 1 byte/cycle). If we normalize the bisection bandwidth by dividing it by the number of nodes (as we did with network bandwidth), the $BW_{\text{Bisection}}$ is 0.25 bytes/cycle/node. Dividing this by γ gives the ideal maximally obtainable network bandwidth of 0.5 bytes/cycle/node.

We can find the efficiency factor, ρ, of the simulated network simply by dividing the measured peak throughput by the ideal throughput. The efficiency factor for the network with fully adaptive routing and four virtual channels is 0.43/(0.25/0.5) = 86%, whereas for the network with deterministic routing and two virtual channels it is 0.37/(0.25/0.5) = 74%. Besides the 12% difference in efficiency between the two, another 14% gain in efficiency might be obtained with even better routing, arbitration, switching, and virtual channel designs.

To put this discussion on routing, arbitration, and switching in perspective, Figure F.20 lists the techniques used in SANs designed for commercial high-performance computers. In addition to being applied to the SANs as shown in the figure, the issues discussed in this section also apply to other interconnect domains: from OCNs to WANs.

F.6 Switch Microarchitecture

Network switches implement the routing, arbitration, and switching functions of switched-media networks. Switches also implement buffer management mechanisms and, in the case of lossless networks, the associated flow control. For some networks, switches also implement part of the network management functions that explore, configure, and reconfigure the network topology in response to boot-up and failures. Here, we reveal the internal structure of network switches by describing a basic switch microarchitecture and various alternatives suitable for different routing, arbitration, and switching techniques presented previously.

Basic Switch Microarchitecture

The internal data path of a switch provides connectivity among the input and output ports. Although a shared bus or a multiported central memory could be used, these solutions are insufficient or too expensive, respectively, when the required aggregate switch bandwidth is high. Most high-performance switches implement an internal crossbar to provide nonblocking connectivity within the switch, thus allowing concurrent connections between multiple input-output port pairs. Buffering of blocked packets can be done using first in, first out (FIFO) or circular queues, which can be implemented as *dynamically allocatable multi-queues* (DAMQs) in static RAM to provide high capacity and flexibility. These queues can be placed at input ports (i.e., *input buffered switch*), output ports (i.e., *output buffered switch*), centrally within the switch (i.e., *centrally buffered switch*), or at both the input and output ports of the switch (i.e., *input-output-buffered switch*). Figure F.21 shows a block diagram of an input-output-buffered switch.

Routing can be implemented using a finite-state machine or forwarding table within the routing control unit of switches. In the former case, the routing information given in the packet header is processed by a finite-state machine that determines the allowed switch output port (or ports if routing is adaptive), according to the routing algorithm. Portions of the routing information in the header are usually

Company	System [network] name	Max. number of nodes [× # CPUs]	Basic network topology	Switch queuing (buffers)	Network routing algorithm	Switch arbitration technique	Network switching technique
Intel	ASCI Red Paragon	4510 [×2]	2D mesh (64 × 64)	Input buffered (1 flit)	Distributed dimension-order routing	2-phased RR, distributed across switch	Wormhole with no virtual channels
IBM	ASCI White SP Power3 [Colony]	512 [×16]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	Input and central buffer with output queuing (8-way speedup)	Source-based LCA adaptive, shortest-path routing, and table-based multicast routing	2-phased RR, centralized and distributed at outputs for bypass paths	Buffered wormhole and virtual cut-through for multicasting, no virtual channels
Intel	Thunder Itanium2 Tiger4 [QsNet^{II}]	1024 [×4]	Fat tree with 8-port bidirectional switches	Input buffered	Source-based LCA adaptive, shortest-path routing	2-phased RR, priority, aging, distributed at output ports	Wormhole with 2 virtual channels
Cray	XT3 [SeaStar]	30,508 [×1]	3D torus (40 × 32 × 24)	Input with staging output	Distributed table-based dimension-order routing	2-phased RR, distributed at output ports	Virtual cut-through with 4 virtual channels
Cray	X1E	1024 [×1]	4-way bristled 2D torus (∼23 × 11) with express links	Input with virtual output queuing	Distributed table-based dimension-order routing	2-phased wavefront (pipelined) global arbiter	Virtual cut-through with 4 virtual channels
IBM	ASC Purple pSeries 575 [Federation]	>1280 [×8]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	Input and central buffer with output queuing (8-way speedup)	Source and distributed table-based LCA adaptive, shortest-path routing, and multicast	2-phased RR, centralized and distributed at outputs for bypass paths	Buffered wormhole and virtual cut-through for multicasting with 8 virtual channels
IBM	Blue Gene/L eServer Solution [Torus Net.]	65,536 [×2]	3D torus (32 × 32 × 64)	Input-output buffered	Distributed, adaptive with bubble escape virtual channel	2-phased SLQ, distributed at input and output	Virtual cut-through with 4 virtual channels

Figure F.20 Routing, arbitration, and switching characteristics of interconnections networks in commercial machines.

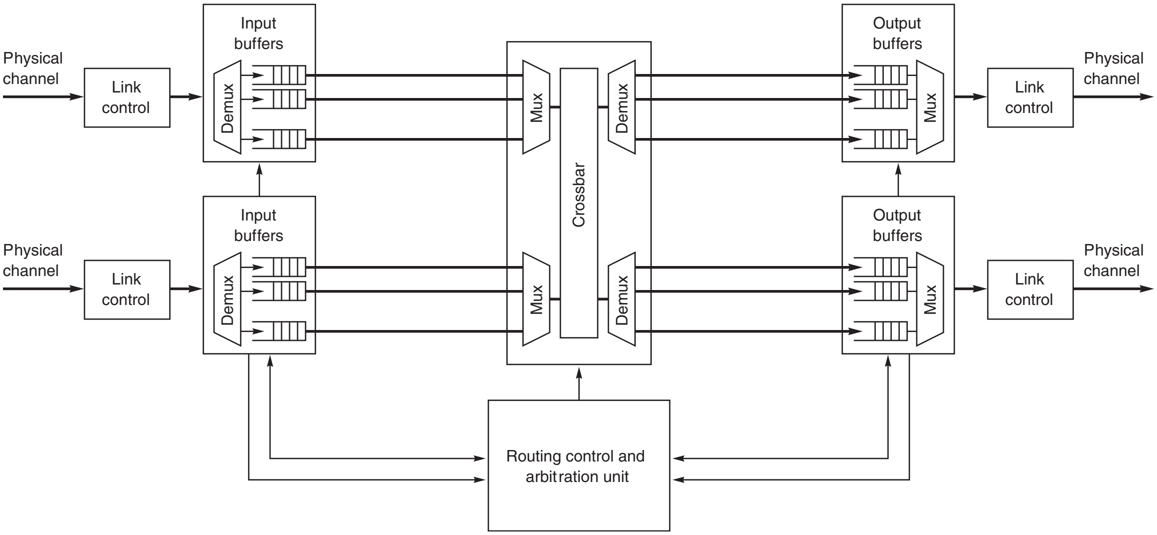

Physical channel
Link control
Input buffers
Demux
Physical channel
Link control
Input buffers
Demux
Mux
Crossbar
Demux
Mux
Demux
Output buffers
Mux
Link control
Physical channel
Output buffers
Mux
Link control
Physical channel
Routing control and arbitration unit

Figure F.21 Basic microarchitectural components of an input-output-buffered switch.

stripped off or modified by the routing control unit after use to simplify processing at the next switch along the path. When routing is implemented using forwarding tables, the routing information given in the packet header is used as an address to access a forwarding table entry that contains the allowed switch output port(s) provided by the routing algorithm. Forwarding tables must be preloaded into the switches at the outset of network operation. Hybrid approaches also exist where the forwarding table is reduced to a small set of routing bits and combined with a small logic block. Those routing bits are used by the routing control unit to know what paths are allowed and decide the output ports the packets need to take. The goal with those approaches is to build flexible yet compact routing control units, eliminating the area and power wastage of a large forwarding table and thus being suitable for OCNs. The routing control unit is usually implemented as a centralized resource, although it could be replicated at every input port so as not to become a bottleneck. Routing is done only once for every packet, and packets typically are large enough to take several cycles to flow through the switch, so a centralized routing control unit rarely becomes a bottleneck. Figure F.21 assumes a centralized routing control unit within the switch.

Arbitration is required when two or more packets concurrently request the same output port, as described in the previous section. Switch arbitration can be implemented in a centralized or distributed way. In the former case, all of the requests and status information are transmitted to the central switch arbitration unit; in the latter case, the arbiter is distributed across the switch, usually among the input and/or output ports. Arbitration may be performed multiple times on packets, and there may be multiple queues associated with each input port, increasing the number of arbitration requests that must be processed. Thus, many implementations use a hierarchical arbitration approach, where arbitration is first performed locally at every input port to select just one request among the corresponding packets and queues, and later arbitration is performed globally to process the requests made by each of the local input port arbiters. Figure F.21 assumes a centralized arbitration unit within the switch.

The basic switch microarchitecture depicted in Figure F.21 functions in the following way. When a packet starts to arrive at a switch input port, the link controller decodes the incoming signal and generates a sequence of bits, possibly deserializing data to adapt them to the width of the internal data path if different from the external link width. Information is also extracted from the packet header or link control signals to determine the queue to which the packet should be buffered. As the packet is being received and buffered (or after the entire packet has been buffered, depending on the switching technique), the header is sent to the routing unit. This unit supplies a request for one or more output ports to the arbitration unit. Arbitration for the requested output port succeeds if the port is free and has enough space to buffer the entire packet or flit, depending on the switching technique. If wormhole switching with virtual channels is implemented, additional arbitration and allocation steps may be required for the transmission of each individual flit. Once the resources are allocated, the packet is transferred across the internal crossbar to the corresponding output buffer and link if no other packets are ahead of it and the link is free. Link-level flow control implemented by the link controller prevents input queue overflow at the neighboring switch on the other end of the link. If virtual channel switching is implemented, several packets may be time-multiplexed across the link on a flit-by-flit basis. As the various input and output ports operate independently, several incoming packets may be processed concurrently in the absence of contention.

Buffer Organizations

As mentioned above, queues can be located at the switch input, output, or both sides. Output-buffered switches have the advantage of completely eliminating *head-of-line blocking*. Head-of-line (HOL) blocking occurs when two or more packets are buffered in a queue, and a blocked packet at the head of the queue blocks other packets in the queue that would otherwise be able to advance if they were at the queue head. This cannot occur in output-buffered switches as all the packets in a given queue have the same status; they require the same output port. However, it may be the case that all the switch input ports simultaneously receive a packet for the same output port. As there are no buffers at the input side, output buffers must be able to store all those incoming packets at the same time. This requires implementing output queues with an internal switch *speedup* of k. That is, output queues must have a write bandwidth k times the link bandwidth, where k is the number of switch ports. This oftentimes is too expensive. Hence, this solution by itself has rarely been implemented in lossless networks. As the probability of concurrently receiving many packets for the same output port is usually small, commercial systems that use output-buffered switches typically implement only moderate switch speedup, dropping packets on rare buffer overflow.

Switches with buffers on the input side are able to receive packets without having any switch speedup; however, HOL blocking can occur within input port queues, as illustrated in Figure F.22(a). This can reduce switch output port utilization to less than 60% even when packet destinations are uniformly distributed. As shown in Figure F.22(b), the use of virtual channels (two in this case) can mitigate HOL blocking but does not eliminate it. A more effective solution is to organize the input queues as *virtual output queues* (VOQs), shown in Figure F.22(c). With this, each input port implements as many queues as there are output ports, thus providing separate buffers for packets destined to different output ports. This is a popular technique widely used in ATM switches and IP routers. The main drawbacks of

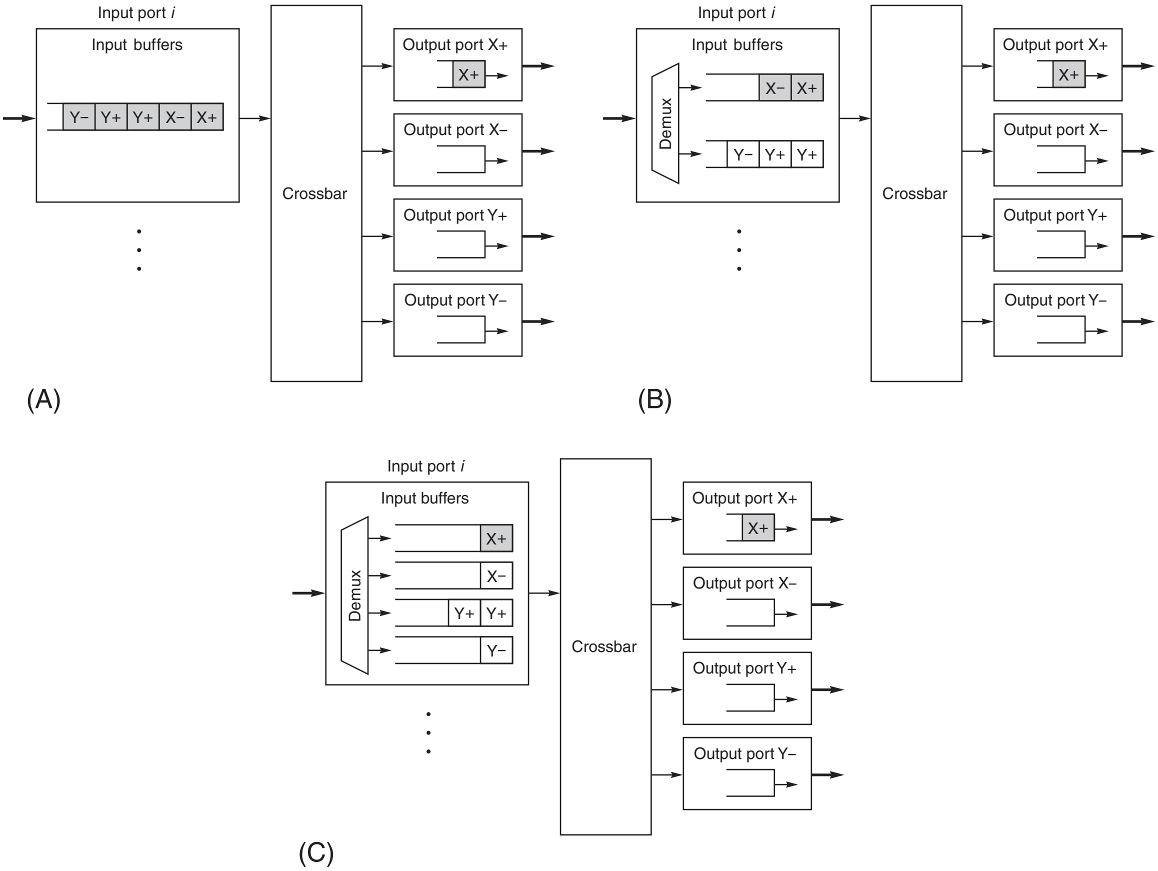

Input port *i*
Input buffers
Y− Y+ Y+ X− X+
Crossbar
Output port X+
X+
Output port X−
Output port Y+
Output port Y−
(A)
Input port *i*
Input buffers
Demux
X− X+
Y− Y+ Y+
Crossbar
Output port X+
X+
Output port X−
Output port Y+
Output port Y−
(B)
Input port *i*
Input buffers
Demux
X+
X−
Y+ Y+
Y−
Crossbar
Output port X+
X+
Output port X−
Output port Y+
Output port Y−
(C)

Figure F.22 (a) Head-of-line blocking in an input buffer, (b) the use of two virtual channels to reduce HOL blocking, and (c) the use of virtual output queuing to eliminate HOL blocking within a switch. The shaded input buffer is the one to which the crossbar is currently allocated. This assumes each input port has only one access port to the switch's internal crossbar.

VOQs, however, are cost and lack of scalability: The number of VOQs grows quadratically with switch ports. Moreover, although VOQs eliminate HOL blocking within a switch, HOL blocking occurring at the network level end-to-end is not solved. Of course, it is possible to design a switch with VOQ support at the network level also—that is, to implement as many queues per switch input port as there are output ports across the entire network—but this is extremely expensive. An alternative is to dynamically assign only a fraction of the queues to store (cache) separately only those packets headed for congested destinations.

Combined input-output-buffered switches minimize HOL blocking when there is sufficient buffer space at the output side to buffer packets, and they minimize the switch speedup required due to buffers being at the input side. This solution has the further benefit of decoupling packet transmission through the internal crossbar of the switch from transmission through the external links. This is especially useful for cut-through switching implementations that use virtual channels, where flit transmissions are time-multiplexed over the links. Many designs used in commercial systems implement input-output-buffered switches.

Routing Algorithm Implementation

It is important to distinguish between the routing algorithm and its implementation. While the routing algorithm describes the rules to forward packets across the network and affects packet latency and network throughput, its implementation affects the delay suffered by packets when reaching a node, the required silicon area, and the power consumption associated with the routing computation. Several techniques have been proposed to pre-compute the routing algorithm and/or hide the routing computation delay. However, significantly less effort has been devoted to reduce silicon area and power consumption without significantly affecting routing flexibility. Both issues have become very important, particularly for OCNs. Many existing designs address these issues by implementing relatively simple routing algorithms, but more sophisticated routing algorithms will likely be needed in the future to deal with increasing manufacturing defects, process variability, and other complications arising from continued technology scaling, as discussed briefly below.

As mentioned in a previous section, depending on where the routing algorithm is computed, two basic forms of routing exist: source and distributed routing. In source routing, the complexity of implementation is moved to the end nodes where paths need to be stored in tables, and the path for a given packet is selected based on the destination end node identifier. In distributed routing, however, the complexity is moved to the switches where, at each hop along the path of a packet, a selection of the output port to take is performed. In distributed routing, two basic implementations exist. The first one consists of using a logic block that implements a fixed routing algorithm for a particular topology. The most common example of such an implementation is dimension-order routing, where dimensions are offset in an established order. Alternatively, distributed routing can be implemented with forwarding tables, where each entry encodes the output port to be used for a particular destination. Therefore, in the worst case, as many entries as destination nodes are required.

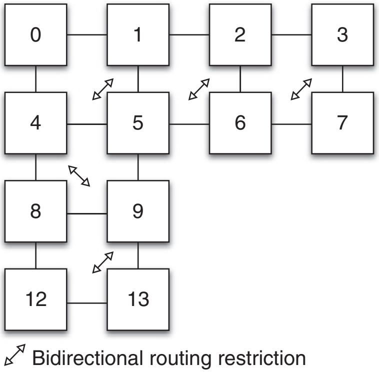
Both methods for implementing distributed routing have their benefits and drawbacks. Logic-based routing features a very short computation delay, usually requires a small silicon area, and has low power consumption. However, logic-based routing needs to be designed with a specific topology in mind and, therefore, is restricted to that topology. Table-based distributed routing is quite flexible and supports any topology and routing algorithm. Simply, tables need to be filled with the proper contents based on the applied routing algorithm (e.g., the up*/down* routing algorithm can be defined for any irregular topology). However, the down side of table-based distributed routing is its non-negligible area and power cost. Also, scalability is problematic in table-based solutions as, in the worst case, a system with N end nodes (and switches) requires as many as N tables each with N entries, thus having quadratic cost.

Depending on the network domain, one solution is more suitable than the other. For instance, in SANs, it is usual to find table-based solutions as is the case with InfiniBand. In other environments, like OCNs, table-based implementations are avoided due to the aforementioned costs in power and silicon area. In such environments, it is more advisable to rely on logic-based implementations. Herein lies some of the challenges OCN designers face: ever continuing technology scaling through device miniaturization leads to increases in the number of manufacturing defects, higher failure rates (either transient or permanent), significant process variations (transistors behaving differently from design specs), the need for different clock frequency and voltage domains, and tight power and energy budgets. All of these challenges translate to the network needing support for heterogeneity. Different—possibly irregular—regions of the network will be created owing to failed components, powered down switches and links, disabled components (due to unacceptable variations in performance) and so on. Hence, heterogeneous systems may emerge from a homogeneous design. In this framework, it is important to efficiently implement routing algorithms designed to provide enough flexibility to address these new challenges.

A well-known solution for providing a certain degree of flexibility while being much more compact than traditional table-based approaches is interval routing [Leeuwen 1987], where a range of destinations is defined for each output port. Although this approach is not flexible enough, it provides a clue on how to address emerging challenges. A more recent approach provides a plausible implementation design point that lies between logic-based implementation (efficiency) and table-based implementation (flexibility). Logic-Based Distributed Routing (LBDR) is a hybrid approach that takes as a reference a regular 2D mesh but allows an irregular network to be derived from it due to changes in topology induced by manufacturing defects, failures, and other anomalies. Due to the faulty, disabled, and powered-down components, regularity is compromised and the dimension-order routing algorithm can no longer be used. To support such topologies, LBDR defines a set of configuration bits at each switch. Four connectivity bits are used at each switch to indicate the connectivity of the switch to the neighbor switches in the topology. Thus, one connectivity bit per port is used. Those connectivity bits are used, for instance, to disable an output port leading to a faulty component. Additionally, eight routing bits are used, two per output port, to define the available routing options. The value of the routing bits is set at power-on and is computed from the routing algorithm to be implemented in the network. Basically, when a routing bit is set, it indicates that a packet can leave the switch through the associated output port and is allowed to perform a certain turn at the next switch. In this respect, LBDR is similar to interval routing, but it defines geographical areas instead of ranges of destinations. Figure F.23 shows an example where a topology-agnostic routing algorithm is implemented with LBDR on an irregular topology. The figure shows the computed configuration bits.

The connectivity and routing bits are used to implement the routing algorithm. For that purpose, a small set of logic gates are used in combination with the configuration bits. Basically, the LBDR approach takes as a reference the initial topology (a 2D mesh), and makes a decision based on the current coordinates of the router, the coordinates of the destination router, and the configuration bits. Figure F.24 shows the required logic, and Figure F.25 shows an example of where a packet is forwarded from its source to its destination with the use of the configuration bits. As can be noticed, routing restrictions are enforced by preventing the use of the west port at switch 10.

LBDR represents a method for efficient routing implementation in OCNs. This mechanism has been recently extended to support non-minimal paths, collective communication operations, and traffic isolation. All of these improvements have been made while maintaining a compact and efficient implementation with the use of a small set of configuration bits. A detailed description of LBDR and its extensions, and the current research on OCNs can be found in Flich [2010].



Router	Cn	Ce	Cw	Cs	Rne	Rnw	Ren	Res	Rwn	Rws	Rse	Rsw
0	0	1	0	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	0
2	0	1	1	1	1	1	1	1	1	1	1	0
3	0	0	1	1	1	1	1	1	1	1	1	0
4	1	1	0	1	1	1	0	1	1	1	0	1
5	1	1	1	1	1	1	0	1	1	1	1	1
6	1	1	1	0	1	1	0	1	1	1	1	1
7	1	0	1	0	1	1	1	1	1	1	1	1
8	1	1	0	1	1	1	1	1	1	1	1	1
9	1	0	1	1	1	1	1	1	0	1	1	0
10	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-
12	1	1	0	0	1	1	0	1	1	1	1	1
13	1	0	1	0	1	1	1	1	1	1	1	1
14	-	-	-	-	-	-	-	-	-	-	-	-
15	-	-	-	-	-	-	-	-	-	-	-	-

Figure F.23 Shown is an example of an irregular network that uses LBDR to implement the routing algorithm. For each router, connectivity and routing bits are defined.

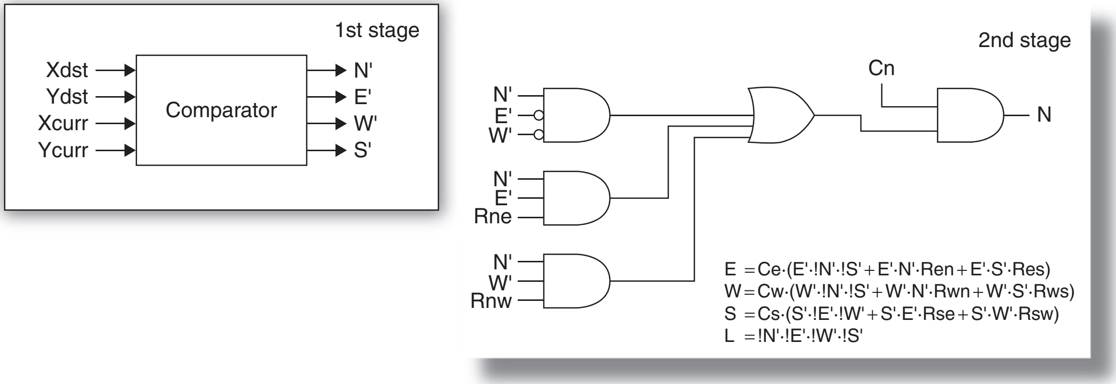

1st stage
Xdst
Ydst
Xcurr
Ycurr
Comparator
N'
E'
W'
S'
2nd stage
Cn
N'
E'
W'
N'
E'
Rne
N'
W'
Rnw
N
E =Ce·(E'·!N'·!S'+E'·N'·Ren+E'·S'·Res)
W=Cw·(W'·!N'·!S'+W'·N'·Rwn+W'·S'·Rws)
S =Cs·(S'·!E'·!W'+S'·E'·Rse+S'·W'·Rsw)
L =!N'·!E'·!W'·!S'

Figure F.24 LBDR logic at each input port of the router.

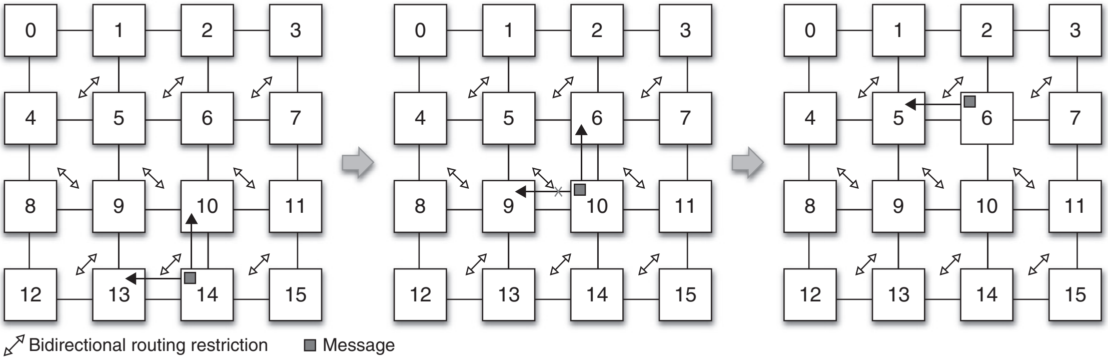
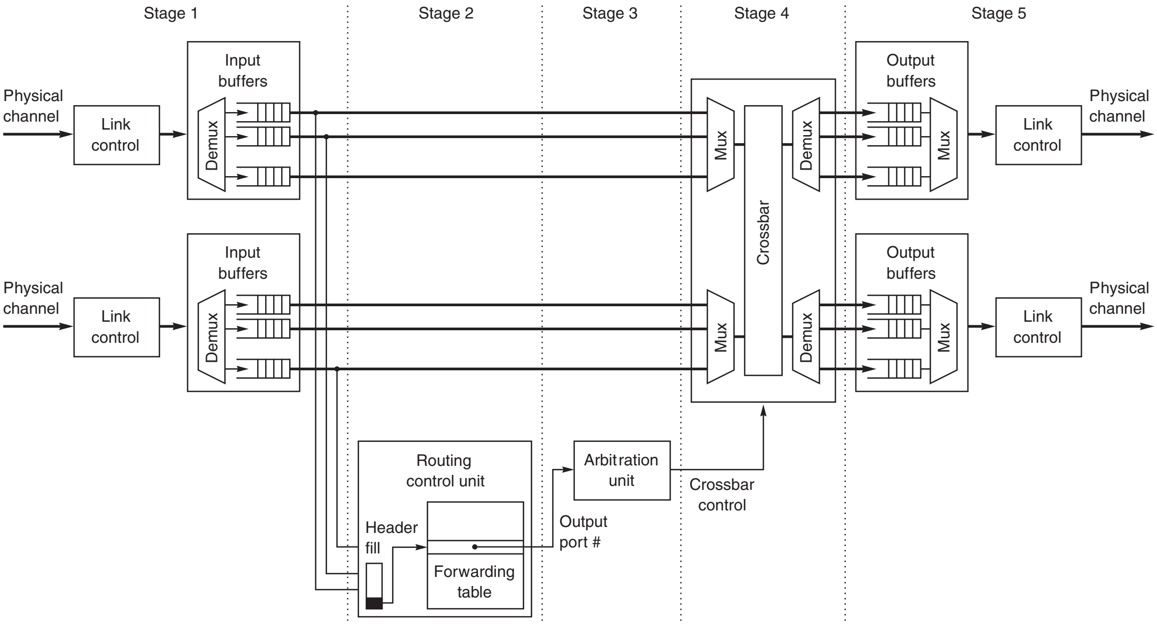

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Bidirectional routing restriction
Message

Figure F.25 Example of routing a message from Router 14 to Router 5 using LBDR at each router.

Pipelining the Switch Microarchitecture

Performance can be enhanced by pipelining the switch microarchitecture. Pipelined processing of packets in a switch has similarities with pipelined execution of instructions in a vector processor. In a vector pipeline, a single instruction indicates what operation to apply to all the vector elements executed in a pipelined way. Similarly, in a switch pipeline, a single packet header indicates how to process all of the internal data path physical transfer units (or *phits*) of a packet, which are processed in a pipelined fashion. Also, as packets at different input ports are independent of each other, they can be processed in parallel similar to the way multiple independent instructions or threads of pipelined instructions can be executed in parallel.

The switch microarchitecture can be pipelined by analyzing the basic functions performed within the switch and organizing them into several stages. Figure F.26 shows a block diagram of a five-stage pipelined organization for the basic switch microarchitecture given in Figure F.21, assuming cut-through switching and the use of a forwarding table to implement routing. After receiving the header portion of the packet in the first stage, the routing information (i.e., destination address) is used in the second stage to look up the allowed routing option(s) in the forwarding table. Concurrent with this, other portions of the packet are received and buffered in the input port queue at the first stage. Arbitration is performed in the third stage. The crossbar is configured to allocate the granted output port for the packet in the fourth stage, and the packet header is buffered in the switch output port and ready for transmission over the external link in the fifth stage. Note that the second and


Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Physical channel
Link control
Input buffers
Demux
Physical channel
Link control
Input buffers
Demux
Mux
Crossbar
Demux
Mux
Demux
Output buffers
Mux
Link control
Physical channel
Output buffers
Mux
Link control
Physical channel
Routing control unit
Header fill
Forwarding table
Output port #
Arbitration unit
Crossbar control

Packet header	IB	RC	SA	ST	OB					
Payload fragment		IB	IB	IB	ST	OB				
Payload fragment			IB	IB	IB	ST	OB			
Payload fragment				IB	IB	IB	ST	OB		

Figure F.26 Pipelined version of the basic input-output-buffered switch. The notation in the figure is as follows: IB is the input link control and buffer stage, RC is the route computation stage, SA is the crossbar switch arbitration stage, ST is the crossbar switch traversal stage, and OB is the output buffer and link control stage. Packet fragments (flits) coming after the header remain in the IB stage until the header is processed and the crossbar switch resources are provided.

third stages are used only by the packet header; the payload and trailer portions of the packet use only three of the stages—those used for data flow-thru once the internal data path of the switch is set up.

A virtual channel switch usually requires an additional stage for virtual channel allocation. Moreover, arbitration is required for every flit before transmission through the crossbar. Finally, depending on the complexity of the routing and arbitration algorithms, several clock cycles may be required for these operations.

Other Switch Microarchitecture Enhancements

As mentioned earlier, internal switch speedup is sometimes implemented to increase switch output port utilization. This speedup is usually implemented by increasing the clock frequency and/or the internal data path width (i.e., phit size) of the switch. An alternative solution consists of implementing several parallel data paths from each input port's set of queues to the output ports. One way of doing this is by increasing the number of crossbar input ports. When implementing several physical queues per input port, this can be achieved by devoting a separate crossbar port to each input queue. For example, the IBM Blue Gene/L implements two crossbar access ports and two read ports per switch input port.

Another way of implementing parallel data paths between input and output ports is to move the buffers to the crossbar crosspoints. This switch architecture is usually referred to as a *buffered crossbar switch*. A buffered crossbar provides independent data paths from each input port to the different output ports, thus making it possible to send up to *k* packets at a time from a given input port to *k* different output ports. By implementing independent crosspoint memories for each input-output port pair, HOL blocking is eliminated at the switch level. Moreover, arbitration is significantly simpler than in other switch architectures. Effectively, each output port can receive packets from only a disjoint subset of the crosspoint memories. Thus, a completely independent arbiter can be implemented at each switch output port, each of those arbiters being very simple.

A buffered crossbar would be the ideal switch architecture if it were not so expensive. The number of crosspoint memories increases quadratically with the number of switch ports, dramatically increasing its cost and reducing its scalability with respect to the basic switch architecture. In addition, each crosspoint memory must be large enough to efficiently implement link-level flow control. To reduce cost, most designers prefer input-buffered or combined input-output-buffered switches enhanced with some of the mechanisms described previously.

F.7 Practical Issues for Commercial Interconnection Networks

There are practical issues in addition to the technical issues described thus far that are important considerations for interconnection networks within certain domains. We mention a few of these below.

Connectivity

The type and number of devices that communicate and their communication requirements affect the complexity of the interconnection network and its protocols. The protocols must target the largest network size and handle the types of anomalous systemwide events that might occur. Among some of the issues are the following: How lightweight should the network interface hardware/software be? Should it attach to the memory network or the I/O network? Should it support cache coherence? If the operating system must get involved for every network transaction, the sending and receiving overhead becomes quite large. If the network interface attaches to the I/O network (PCI-Express or HyperTransport interconnect), the injection and reception bandwidth will be limited to that of the I/O network. This is the case for the Cray XT3 SeaStar, Intel Thunder Tiger 4 QsNet^{II}, and many other supercomputer and cluster networks. To support coherence, the sender may have to flush the cache before each send, and the receiver may have to flush its cache before each receive to prevent the stale-data problem. Such flushes further increase sending and receiving overhead, often causing the network interface to be the network bottleneck.

Computer systems typically have a multiplicity of interconnects with different functions and cost-performance objectives. For example, processor-memory interconnects usually provide higher bandwidth and lower latency than I/O interconnects and are more likely to support cache coherence, but they are less likely to follow or become standards. Personal computers typically have a processormemory interconnect and an I/O interconnect (e.g., PCI-X 2.0, PCIe or Hyper-Transport) designed to connect both fast and slow devices (e.g., USB 2.0, Gigabit Ethernet LAN, Firewire 800). The Blue Gene/L supercomputer uses five interconnection networks, only one of which is the 3D torus used for most of the interprocessor application traffic. The others include a tree-based collective communication network for broadcast and multicast; a tree-based barrier network for combining results (scatter, gather); a control network for diagnostics, debugging, and initialization; and a Gigabit Ethernet network for I/O between the nodes and disk. The University of Texas at Austin's TRIPS Edge processor has eight specialized on-chip networks—some with bidirectional channels as wide as 128 bits and some with 168 bits in each direction—to interconnect the 106 heterogeneous tiles composing the two processor cores with L2 on-chip cache. It also has a chip-to-chip switched network to interconnect multiple chips in a multiprocessor configuration. Two of the on-chip networks are switched networks: One is used for operand transport and the other is used for on-chip memory communication. The others are essentially fan-out trees or recombination dedicated link networks used for status and control. The portion of chip area allocated to the interconnect is substantial, with five of the seven metal layers used for global network wiring.

Standardization: Cross-Company Interoperability

Standards are useful in many places in computer design, including interconnection networks. Advantages of successful standards include low cost and stability.

The customer has many vendors to choose from, which keeps price close to cost due to competition. It makes the viability of the interconnection independent of the stability of a single company. Components designed for a standard interconnection may also have a larger market, and this higher volume can reduce the vendors' costs, further benefiting the customer. Finally, a standard allows many companies to build products with interfaces to the standard, so the customer does not have to wait for a single company to develop interfaces to all the products of interest.

One drawback of standards is the time it takes for committees and special-interest groups to agree on the definition of standards, which is a problem when technology is changing rapidly. Another problem is *when* to standardize: On the one hand, designers would like to have a standard before anything is built; on the other hand, it would be better if something were built before standardization to avoid legislating useless features or omitting important ones. When done too early, it is often done entirely by committee, which is like asking all of the chefs in France to prepare a single dish of food—masterpieces are rarely served. Standards can also suppress innovation at that level, since standards fix the interfaces—at least until the next version of the standards surface, which can be every few years or longer. More often, we are seeing consortiums of companies getting together to define and agree on technology that serve as "*de facto*" industry standards. This was the case for InfiniBand.

LANs and WANs use standards and interoperate effectively. WANs involve many types of companies and must connect to many brands of computers, so it is difficult to imagine a proprietary WAN ever being successful. The ubiquitous nature of the Ethernet shows the popularity of standards for LANs as well as WANs, and it seems unlikely that many customers would tie the viability of their LAN to the stability of a single company. Some SANs are standardized such as Fibre Channel, but most are proprietary. OCNs for the most part are proprietary designs, with a few gaining widespread commercial use in system-on-chip (SoC) applications, such as IBM's CoreConnect and ARM's AMBA.

Congestion Management

Congestion arises when too many packets try to use the same link or set of links. This leads to a situation in which the bandwidth required exceeds the bandwidth supplied. Congestion by itself does not degrade network performance: simply, the congested links are running at their maximum capacity. Performance degradation occurs in the presence of HOL blocking where, as a consequence of packets going to noncongested destinations getting blocked by packets going to congested destinations, some link bandwidth is wasted and network throughput drops, as illustrated in the example given at the end of Section F.4. *Congestion control* refers to schemes that reduce traffic when the collective traffic of all nodes is too large for the network to handle.

One advantage of a circuit-switched network is that, once a circuit is established, it ensures that there is sufficient bandwidth to deliver all the information sent along that circuit. Interconnection bandwidth is reserved as circuits are established, and if the network is full, no more circuits can be established. Other switching techniques generally do not reserve interconnect bandwidth in advance, so the interconnection network can become clogged with too many packets. Just as with poor rush-hour commuters, a traffic jam of packets increases packet latency and, in extreme cases, fewer packets per second get delivered by the interconnect. In order to handle congestion in packet-switched networks, some form of *congestion management* must be implemented. The two kinds of mechanisms used are those that control congestion and those that eliminate the performance degradation introduced by congestion.

There are three basic schemes used for congestion control in interconnection networks, each with its own weaknesses: packet discarding, flow control, and choke packets. The simplest scheme is *packet discarding*, which we discussed briefly in Section F.2. If a packet arrives at a switch and there is no room in the buffer, the packet is discarded. This scheme relies on higher-level software that handles errors in transmission to resend lost packets. This leads to significant bandwidth wastage due to (re)transmitted packets that are later discarded and, therefore, is typically used only in lossy networks like the Internet.

The second scheme relies on *flow control*, also discussed previously. When buffers become full, link-level flow control provides feedback that prevents the transmission of additional packets. This *backpressure* feedback rapidly propagates backward until it reaches the sender(s) of the packets producing congestion, forcing a reduction in the injection rate of packets into the network. The main drawbacks of this scheme are that sources become aware of congestion too late when the network is already congested, and nothing is done to alleviate congestion. Backpressure flow control is common in lossless networks like SANs used in supercomputers and enterprise systems.

A more elaborate way of using flow control is by implementing it directly between the sender and the receiver end nodes, generically called *end-to-end flow control*. *Windowing* is one version of end-to-end credit-based flow control where the window size should be large enough to efficiently pipeline packets through the network. The goal of the window is to limit the number of unacknowledged packets, thus bounding the contribution of each source to congestion, should it arise. The TCP protocol uses a sliding window. Note that end-to-end flow control describes the interaction between just two nodes of the interconnection network, not the entire interconnection network between all end nodes. Hence, flow control helps congestion control, but it is not a global solution.

Choke packets are used in the third scheme, which is built upon the premise that traffic injection should be throttled only when congestion exists across the network. The idea is for each switch to see how busy it is and to enter into a warning state when it passes a threshold. Each packet received by a switch in the warning state is sent back to the source via a choke packet that includes the intended destination. The source is expected to reduce traffic to that destination by a fixed percentage. Since it likely will have already sent other packets along that path, the source node waits for all the packets in transit to be returned before acting on the choke packets. In this scheme, congestion is controlled by reducing the packet injection rate until traffic reduces, just as metering lights that guard on-ramps control the rate of cars entering a freeway. This scheme works efficiently when the feedback delay is short. When congestion notification takes a long time, usually due to long time of flight, this congestion control scheme may become unstable—reacting too slowly or producing oscillations in packet injection rate, both of which lead to poor network bandwidth utilization.

An alternative to congestion control consists of eliminating the negative consequences of congestion. This can be done by eliminating HOL blocking at every switch in the network as discussed previously. Virtual output queues can be used for this purpose; however, it would be necessary to implement as many queues at every switch input port as devices attached to the network. This solution is very expensive, and not scalable at all. Fortunately, it is possible to achieve good results by dynamically assigning a few set-aside queues to store only the congested packets that travel through some hot-spot regions of the network, very much like caches are intended to store only the more frequently accessed memory locations. This strategy is referred to as *regional explicit congestion notification* (RECN).

Fault Tolerance

The probability of system failures increases as transistor integration density and the number of devices in the system increases. Consequently, system reliability and availability have become major concerns and will be even more important in future systems with the proliferation of interconnected devices. A practical issue arises, therefore, as to whether or not the interconnection network relies on all the devices being operational in order for the network to work properly. Since software failures are generally much more frequent than hardware failures, another question surfaces as to whether a software crash on a single device can prevent the rest of the devices from communicating. Although some hardware designers try to build fault-free networks, in practice, it is only a question of the rate of failures, not whether they can be prevented. Thus, the communication subsystem must have mechanisms for dealing with faults when—not if—they occur.

There are two main kinds of failure in an interconnection network: *transient* and *permanent*. Transient failures are usually produced by electromagnetic interference and can be detected and corrected using the techniques described in Section F.2. Oftentimes, these can be dealt with simply by retransmitting the packet either at the link level or end-to-end. Permanent failures occur when some component stops working within specifications. Typically, these are produced by overheating, overbiasing, overuse, aging, and so on and cannot be recovered from simply by retransmitting packets with the help of some higher-layer software protocol. Either an alternative physical path must exist in the network and be supplied by the routing algorithm to circumvent the fault or the network will be crippled, unable to deliver packets whose only paths are through faulty resources.

Three major categories of techniques are used to deal with permanent failures: *resource sparing, fault-tolerant routing*, and *network reconfiguration*. In the first technique, faulty resources are switched off or bypassed, and some spare resources are switched in to replace the faulty ones. As an example, the ServerNet interconnection network is designed with two identical switch fabrics, only one of which is usable at any given time. In case of failure in one fabric, the other is used. This technique can also be implemented without switching in spare resources, leading to a degraded mode of operation after a failure. The IBM Blue Gene/L supercomputer, for instance, has the facility to bypass failed network resources while retaining its base topological structure and routing algorithm. The main drawback of this technique is the relatively large number of healthy resources (e.g., midplane node boards) that may need to be switched off after a failure in order to retain the base topological structure (e.g., a 3D torus).

Fault-tolerant routing, on the other hand, takes advantage of the multiple paths already existing in the network topology to route messages in the presence of failures without requiring spare resources. Alternative paths for each supported fault combination are identified at design time and incorporated into the routing algorithm. When a fault is detected, a suitable alternative path is used. The main difficulty when using this technique is guaranteeing that the routing algorithm will remain deadlock-free when using the alternative paths, given that arbitrary fault patterns may occur. This is especially difficult in direct networks whose regularity can be compromised by the fault pattern. The Cray T3E is an example system that successfully applies this technique on its 3D torus direct network. There are many examples of this technique in systems using indirect networks, such as with the bidirectional multistage networks in the ASCI White and ASC Purple. Those networks provide multiple minimal paths between end nodes and, inherently, have no routing deadlock problems (see Section F.5). In these networks, alternative paths are selected at the source node in case of failure.

Network reconfiguration is yet another, more general technique to handle voluntary and involuntary changes in the network topology due either to failures or to some other cause. In order for the network to be reconfigured, the nonfaulty portions of the topology must first be discovered, followed by computation of the new routing tables and distribution of the routing tables to the corresponding network locations (i.e., switches and/or end node devices). Network reconfiguration requires the use of programmable switches and/or network interfaces, depending on how routing is performed. It may also make use of generic routing algorithms (e.g., up*/down* routing) that can be configured for all the possible network topologies that may result after faults. This strategy relieves the designer from having to supply alternative paths for each possible fault combination at design time. Programmable network components provide a high degree of flexibility but at the expense of higher cost and latency. Most standard and proprietary interconnection networks for clusters and SANs—including Myrinet, Quadrics, InfiniBand, Advanced Switching, and Fibre Channel—incorporate software for (re)configuring the network routing in accordance with the prevailing topology.

Another practical issue ties to node failure tolerance. If an interconnection network can survive a failure, can it also continue operation while a new node is added to or removed from the network, usually referred to as *hot swapping*? If not, each addition or removal of a new node disables the interconnection network, which is impractical for WANs and LANs and is usually intolerable for most SANs. Online system expansion requires hot swapping, so most networks allow for it. Hot swapping is usually supported by implementing *dynamic network reconfiguration*, in which the network is reconfigured without having to stop user traffic. The main difficulty with this is guaranteeing deadlock-free routing while routing tables for switches and/or end node devices are dynamically and asynchronously updated as more than one routing algorithm may be alive (and, perhaps, clashing) in the network at the same time. Most WANs solve this problem by dropping packets whenever required, but dynamic network reconfiguration is much more complex in lossless networks. Several theories and practical techniques have recently been developed to address this problem efficiently.

Example Figure F.27 shows the number of failures of 58 desktop computers on a local area network for a period of just over one year. Suppose that one local area network is based on a network that requires all machines to be operational for the interconnection network to send data; if a node crashes, it cannot accept messages, so the interconnection becomes choked with data waiting to be delivered. An alternative is the traditional local area network, which can operate in the presence of node failures; the interconnection simply discards messages for a node that decides not to accept them. Assuming that you need to have both your workstation and the connecting LAN to get your work done, how much greater are your chances of being prevented from getting your work done using the failure-intolerant LAN versus traditional LANs? Assume the downtime for a crash is less than 30 minutes. Calculate using the one-hour intervals from this figure.

Answer Assuming the numbers for Figure F.27, the percentage of hours that you can't get your work done using the failure-intolerant network is

$$\frac{\text{Intervals with failures}}{\text{Total intervals}} = \frac{\text{Total intervals} - \text{Intervals with no failures}}{\text{Total intervals}}$$

$$= \frac{8974 - 8605}{8974} = \frac{369}{8974} = 4.1\%$$

The percentage of hours that you can't get your work done using the traditional network is just the time your workstation has crashed. If these failures are equally distributed among workstations, the percentage is

$$\frac{\text{Failures/Machines}}{\text{Total intervals}} = \frac{654/58}{8974} = \frac{11.28}{8974} = 0.13\%$$

Hence, you are more than 30 times more likely to be prevented from getting your work done with the failure-intolerant LAN than with the traditional LAN, according to the failure statistics in Figure F.27. Stated alternatively, the person responsible for maintaining the LAN would receive a 30-fold increase in phone calls from irate users!

Failed machines per time interval	One-hour intervals with number of failed machines in first column	Total failures per one-hour interval	One-day intervals with number of failed machines in first column	Total failures per one-day interval
0	8605	0	184	0
1	264	264	105	105
2	50	100	35	70
3	25	75	11	33
4	10	40	6	24
5	7	35	9	45
6	3	18	6	36
7	1	7	4	28
8	1	8	4	32
9	2	18	2	18
10	2	20		
11	1	11	2	22
12			1	12
17	1	17		
20	1	20		
21	1	21	1	21
31			1	31
38			1	38
58			1	58
Total	**8974**	**654**	**373**	**573**

Figure F.27 Measurement of reboots of 58 DECstation 5000 s running Ultrix over a 373-day period. These reboots are distributed into time intervals of one hour and one day. The first column sorts the intervals according to the number of machines that failed in that interval. The next two columns concern one-hour intervals, and the last two columns concern one-day intervals. The second and fourth columns show the number of intervals for each number of failed machines. The third and fifth columns are just the product of the number of failed machines and the number of intervals. For example, there were 50 occurrences of one-hour intervals with 2 failed machines, for a total of 100 failed machines, and there were 35 days with 2 failed machines, for a total of 70 failures. As we would expect, the number of failures per interval changes with the size of the interval. For example, the day with 31 failures might include one hour with 11 failures and one hour with 20 failures. The last row shows the total number of each column; the number of failures doesn't agree because multiple reboots of the same machine in the same interval do not result in separate entries. (Randy Wang of the University of California–Berkeley collected these data.)

F.8 Examples of Interconnection Networks

To further provide mass to the concepts described in the previous sections, we look at five example networks from the four interconnection network domains considered in this appendix. In addition to one for each of the OCN, LAN, and WAN areas, we look at two examples from the SAN area: one for system area networks and one for system/storage area networks. The first two examples are proprietary networks used in high-performance systems; the latter three examples are network standards widely used in commercial systems.

On-Chip Network: Intel Single-Chip Cloud Computer

With continued increases in transistor integration as predicted by Moore's law, processor designers are under the gun to find ways of combating chip-crossing wire delay and other problems associated with deep submicron technology scaling. Multicore microarchitectures have gained popularity, given their advantages of simplicity, modularity, and ability to exploit parallelism beyond that which can be achieved through aggressive pipelining and multiple instruction/data issuing on a single core. No matter whether the processor consists of a single core or multiple cores, higher and higher demands are being placed on intrachip communication bandwidth to keep pace—not to mention interchip bandwidth. This has spurred a great amount of interest in OCN designs that efficiently support communication of instructions, register operands, memory, and I/O data within and between processor cores both on and off the chip. Here we focus on one such on-chip network: The Intel Single-chip Cloud Computer prototype.

The Single-chip Cloud Computer (SCC) is a prototype chip multiprocessor with 48 Intel IA-32 architecture cores. Cores are laid out (see Figure F.28) on a network with a 2D mesh topology (6 × 4). The network connects 24 tiles, 4 on-die memory controllers, a voltage regulator controller (VRC), and an external system interface controller (SIF). In each tile two cores are connected to a router. The four memory controllers are connected at the boundaries of the mesh, two on each side, while the VRC and SIF controllers are connected at the bottom border of the mesh.

Each memory controller can address two DDR3 DIMMS, each up to 8 GB of memory, thus resulting in a maximum of 64 GB of memory. The VRC controller allows any core or the system interface to adjust the voltage in any of the six predefined regions configuring the network (two 2-tile regions). The clock can also be adjusted at a finer granularity with each tile having its own operating frequency. These regions can be turned off or scaled down for large power savings. This method allows full application control of the power state of the cores. Indeed, applications have an API available to define the voltage and the frequency of each region. The SIF controller is used to communicate the network from outside the chip.

Each of the tiles includes two processor cores (P54C-based IA) with associated L1 16 KB data cache and 16 KB instruction cache and a 256 KB L2 cache (with the associated controller), a 5-port router, traffic generator (for testing purposes only), a mesh interface unit (MIU) handling all message passing requests, memory look-up tables (with configuration registers to set the mapping of a core's physical addresses to the extended memory map of the system), a message-passing buffer, and circuitry for the clock generation and synchronization for crossing asynchronous boundaries.

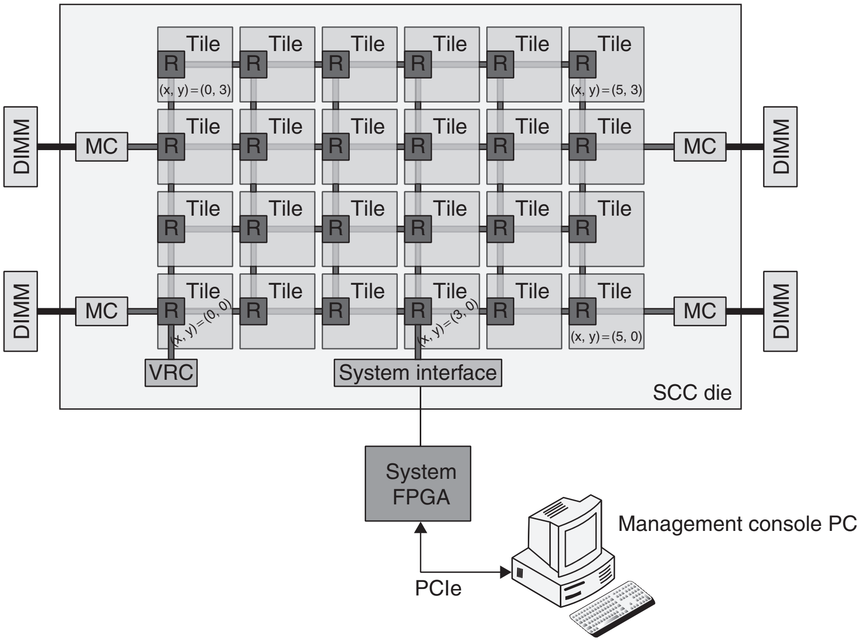


Figure F.28 SCC Top-level architecture. From Howard, J. et al., *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 58–59.

Focusing on the OCN, the MIU unit is in charge of interfacing the cores to the network, including the packetization and de-packetization of large messages; command translation and address decoding/lookup; link-level flow control and credit management; and arbiter decisions following a round-robin scheme. A credit-based flow control mechanism is used together with virtual cut-through switching (thus making it necessary to split long messages into packets). The routers are connected in a 2D mesh layout, each on its own power supply and clock source. Links connecting routers have 16B + 2B side bands running at 2 GHz. Zero-load latency is set to 4 cycles, including link traversal. Eight virtual channels are used for performance (6 VCs) and protocol-level deadlock handling (2 VCs). A message-level arbitration is implemented by a wrapped wave-front arbiter. The dimension-order XY routing algorithm is used and pre-computation of the output port is performed at every router.

Besides the tiles having regions defined for voltage and frequency, the network (made of routers and links) has its own single region. Thus, all the network components run at the same speed and use the same power supply. An asynchronous clock transition is required between the router and the tile.

One of the distinctive features of the SCC architecture is the support for a messaging-based communication protocol rather than hardware cache-coherent memory for inter-core communication. Message passing buffers are located on every router and APIs are provided to take full control of MPI structures. Cache coherency can be implemented by software.

The SCC router represents a significant improvement over the Teraflops processor chip in the implementation of a 2D on-chip interconnect. Contrasted with the 2D mesh implemented in the Teraflops processor, this implementation is tuned for a wider data path in a multiprocessor interconnect and is more latency, area, and power optimized for such a width. It targets a lower 2-GHz frequency of operation compared to the 5 GHz of its predecessor Teraflops processor, yet with a higher-performance interconnect architecture.

System Area Network: IBM Blue Gene/L 3D Torus Network

The IBM BlueGene/L was the largest-scaled, highest-performing computer system in the world in 2005, according to *www.top500.org*. With 65,536 dual-processor compute nodes and 1024 I/O nodes, this 360 TFLOPS (peak) supercomputer has a system footprint of approximately 2500 square feet. Both processors at each node can be used for computation and can handle their own communication protocol processing in virtual mode or, alternatively, one of the processors can be used for computation and the other for network interface processing. Packets range in size from 32 bytes to a maximum of 256 bytes, and 8 bytes are used for the header. The header includes routing, virtual channel, link-level flow control, packet size, and other such information, along with 1 byte for CRC to protect the header. Three bytes are used for CRC at the packet level, and 1 byte serves as a valid indicator.

The main interconnection network is a proprietary 32 × 32 × 64 3D torus SAN that interconnects all 64 K nodes. Each node switch has six 350 MB/sec bidirectional links to neighboring torus nodes, an injection bandwidth of 612.5 MB/sec from the two node processors, and a reception bandwidth of 1050 MB/sec to the two node processors. The reception bandwidth from the network equals the inbound bandwidth across all switch ports, which prevents reception links from bottlenecking network performance. Multiple packets can be sunk concurrently at each destination node because of the higher reception link bandwidth.

Two nodes are implemented on a 2 × 1 × 1 compute card, 16 compute cards and 2 I/O cards are implemented on a 4 × 4 × 2 node board, 16 node boards are implemented on an 8 × 8 × 8 midplane, and 2 midplanes form a 1024-node rack with physical dimensions of 0.9 × 0.9 × 1.9 cubic meters. Links have a maximum physical length of 8.6 meters, thus enabling efficient link-level flow control with reasonably low buffering requirements. Low latency is achieved by implementing virtual cut-through switching, distributing arbitration at switch input and output ports, and precomputing the current routing path at the previous switch using a finite-state machine so that part of the routing delay is removed from the critical path in switches. High effective bandwidth is achieved using input-buffered switches with dual read ports, virtual cut-through switching with four virtual channels, and fully adaptive deadlock-free routing based on bubble flow control.

A key feature in networks of this size is fault tolerance. Failure rate is reduced by using a relatively low link clock frequency of 700 MHz (same as processor clock) on which both edges of the clock are used (i.e., 1.4 Gbps or 175 MB/sec transfer rate is supported for each bit-serial network link in each direction), but failures may still occur in the network. In case of failure, the midplane node boards containing the fault(s) are switched off and bypassed to isolate the fault, and computation resumes from the last checkpoint. Bypassing is done using separate bypass switch boards associated with each midplane that are additional to the set of torus node boards. Each bypass switch board can be configured to connect either to the corresponding links in the midplane node boards or to the next bypass board, effectively removing the corresponding set of midplane node boards. Although the number of processing nodes is reduced to some degree in some network dimensions, the machine retains its topological structure and routing algorithm.

Some collective communication operations such as barrier synchronization, broadcast/multicast, reduction, and so on are not performed well on the 3D torus as the network would be flooded with traffic. To remedy this, two separate tree networks with higher per-link bandwidth are used to implement collective and combining operations more efficiently. In addition to providing support for efficient synchronization and broadcast/multicast, hardware is used to perform some arithmetic reduction operations in an efficient way (e.g., to compute the sum or the maximum value of a set of values, one from each processing node). In addition to the 3D torus and the two tree networks, the Blue Gene/L implements an I/O Gigabit Ethernet network and a control system Fast Ethernet network of lower bandwidth to provide for parallel I/O, configuration, debugging, and maintenance.

System/Storage Area Network: InfiniBand

InfiniBand is an industrywide *de facto* networking standard developed in October 2000 by a consortium of companies belonging to the InfiniBand Trade Association. InfiniBand can be used as a system area network for interprocessor communication or as a storage area network for server I/O. It is a switch-based interconnect technology that provides flexibility in the topology, routing algorithm, and arbitration technique implemented by vendors and users. InfiniBand supports data transmission rates of 2 to 120 Gbp/link per direction across distances of 300 meters. It uses cut-through switching, 16 virtual channels and service levels, credit-based link-level flow control, and weighted round-robin fair scheduling and implements programmable forwarding tables. It also includes features useful for increasing reliability and system availability, such as communication subnet management, end-to-end path establishment, and virtual destination naming.

Institution and processor [network] name	Year built	Number of network ports [cores or tiles + other ports]	Basic network topology	# of data bits per link per direction	Link bandwidth [link clock speed]	Routing; arbitration; switching	# of chip metal layers; flow control; #virtual channels
MIT Raw [General Dynamic Network]	2002	16 ports [16 tiles]	2D mesh (4 × 4)	32 bits	0.9 GB/sec [225 MHz, clocked at proc speed]	XY DOR with request-reply deadlock recovery; RR arbitration; wormhole	6 layers; credit-based no virtual channels
IBM Power5	2004	7 ports [2 PE cores + 5 other ports]	Crossbar	256 bits Inst fetch; 64 bits for stores; 256 bits LDs	[1.9 GHz, clocked at proc speed]	Shortest-path; nonblocking; circuit switch	7 layers; handshaking; no virtual channels
U.T. Austin TRIP Edge [Operand Network]	2005	25 ports [25 execution unit tiles]	2D mesh (5 × 5)	110 bits	5.86 GB/sec [533 MHz clock scaled by 80%]	YX DOR; distributed RR arbitration; wormhole	7 layers; on/off flow control; no virtual channels
U.T. Austin TRIP Edge [On-Chip Network]	2005	40 ports [16 L2 tiles + 24 network interface tile]	2D mesh (10 × 4)	128 bits	6.8 GB/sec [533 MHz clock scaled by 80%]	YX DOR; distributed RR arbitration; VCT switched	7 layers; credit-based flow control; 4 virtual channels
Sony, IBM, Toshiba Cell BE [Element Interconnect Bus]	2005	12 ports [1 PPE and 8 SPEs + 3 other ports for memory, I/O interface]	Ring (4 total, 2 in each direction)	128 bits data (+16 bits tag)	25.6 GB/sec [1.6 GHz, clocked at half the proc speed]	Shortest-path; tree-based RR arbitration (centralized); pipelined circuit switch	8 layers; credit-based flow control; no virtual channels
Sun UltraSPARC T1 processor	2005	Up to 13 ports [8 PE cores + 4 L2 banks + 1 shared I/O]	Crossbar	128 bits both for the 8 cores and the 4 L2 banks	19.2 GB/sec [1.2 GHz, clocked at proc speed]	Shortest-path; age-based arbitration; VCT switched	9 layers; handshaking; no virtual channels

Figure F.29 Characteristics of on-chip networks implemented in recent research and commercial processors. Some processors implement multiple on-chip networks (not all shown)—for example, two in the MIT Raw and eight in the TRIP Edge.

Figure F.30 shows the packet format for InfiniBand juxtaposed with two other network standards from the LAN and WAN areas. Figure F.31 compares various characteristics of the InfiniBand standard with two proprietary system area networks widely used in research and commercial high-performance computer systems.

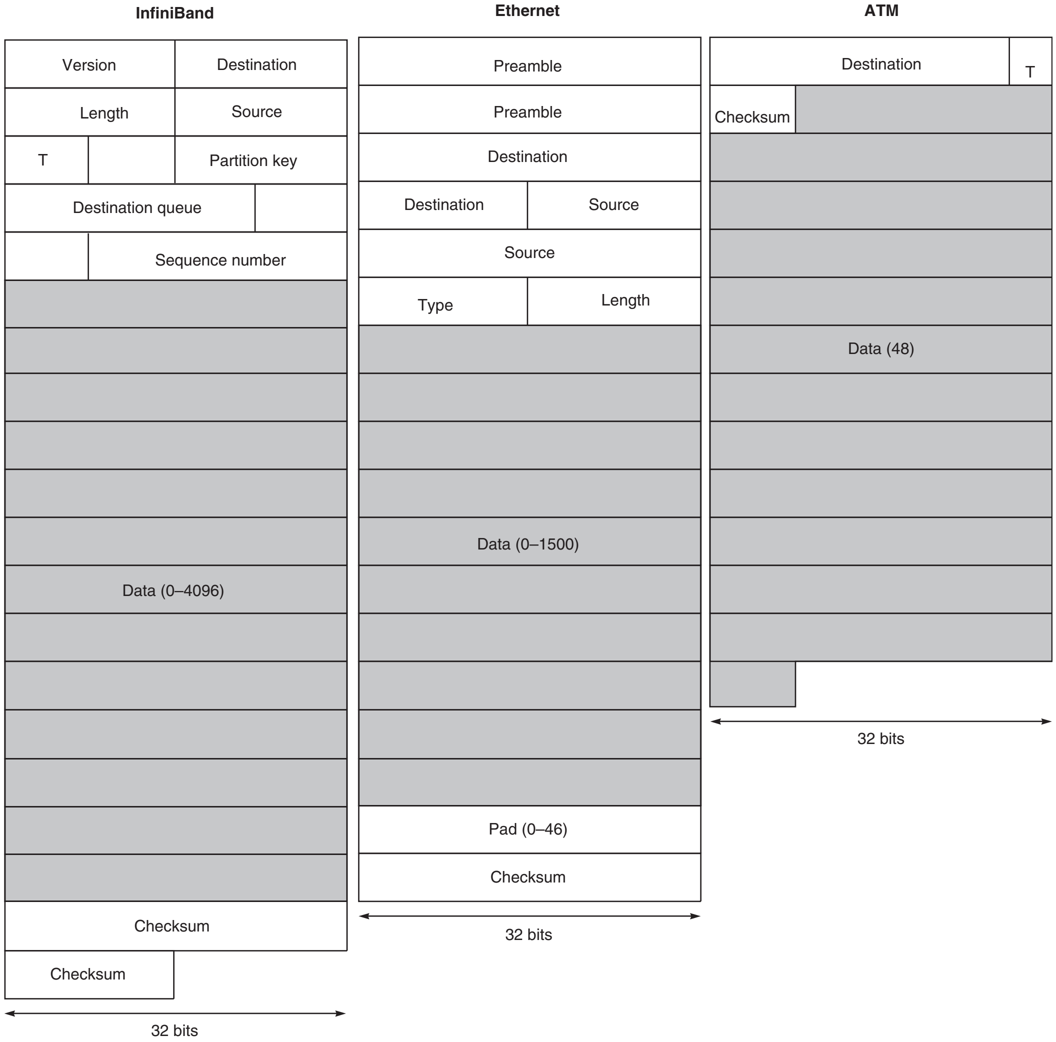

InfiniBand
Version
Destination
Length
Source
T
Partition key
Destination queue
Sequence number
Data (0–4096)
Checksum
Checksum
32 bits
Ethernet
Preamble
Preamble
Destination
Destination
Source
Source
Type
Length
Data (0–1500)
Pad (0–46)
Checksum
32 bits
ATM
Destination
T
Checksum
Data (48)
32 bits

Figure F.30 Packet format for InfiniBand, Ethernet, and ATM. ATM calls their messages "cells" instead of packets, so the proper name is ATM cell format. The width of each drawing is 32 bits. All three formats have destination addressing fields, encoded differently for each situation. All three also have a checksum field to catch transmission errors, although the ATM checksum field is calculated only over the header; ATM relies on higher-level protocols to catch errors in the data. Both InfiniBand and Ethernet have a length field, since the packets hold a variable amount of data, with the former counted in 32-bit words and the latter in bytes. InfiniBand and ATM headers have a type field (T) that gives the type of packet. The remaining Ethernet fields are a preamble to allow the receiver to recover the clock from the self-clocking code used on the Ethernet, the source address, and a pad field to make sure the smallest packet is 64 bytes (including the header). InfiniBand includes a version field for protocol version, a sequence number to allow in-order delivery, a field to select the destination queue, and a partition key field. Infiniband has many more small fields not shown and many other packet formats; above is a simplified view. ATM's short, fixed packet is a good match to real-time demand of digital voice.

Network name [vendors]	**Used in top 10 supercomputer clusters (2005)**	**Number of nodes**	**Basic network topology**	**Raw link bidirectional BW**	**Routing algorithm**	**Arbitration technique**	**Switching technique; flow control**
InfiniBand [Mellanox, Voltair]	SGI Altrix and Dell Poweredge Thunderbird	>Millions (2^{128} GUID addresses, like IPv6)	Completely configurable (arbitrary)	4–240 Gbps	Arbitrary (table-driven), typically up*/down*	Weighted RR fair scheduling (2-level priority)	Cut-through, 16 virtual channels (15 for data); credit-based
Myrinet-2000 [Myricom]	Barcelona Supercomputer Center in Spain	8192 nodes	Bidirectional MIN with 16-port bidirectional switches (Clos net.)	4 Gbps	Source-based dispersive (adaptive) minimal routing	Round-robin arbitration	Cut-through switching with no virtual channels; Xon/Xoff flow control
QsNet^{II} [Quadrics]	Intel Thunder Itanium2 Tiger4	>Tens of thousands	Fat tree with 8-port bidirectional switches	21.3 Gbps	Source-based LCA adaptive shortest-path routing	2-phased RR, priority, aging, distributed at output ports	Wormhole with 2 virtual channels; credit-based

Figure F.31 Characteristics of system area networks implemented in various top 10 supercomputer clusters in 2005.

InfiniBand offers two basic mechanisms to support user-level communication: send/receive and remote DMA (RDMA). With send/receive, the receiver has to explicitly post a receive buffer (i.e., allocate space in its channel adapter network interface) before the sender can transmit data. With RDMA, the sender can remotely DMA data directly into the receiver device's memory. For example, for a nominal packet size of 4 bytes measured on a Mellanox MHEA28-XT channel adapter connected to a 3.4 GHz Intel Xeon host device, sending and receiving overhead is 0.946 and 1.423 μs, respectively, for the send/receive mechanism, whereas it is 0.910 and 0.323 μs, respectively, for the RDMA mechanism.

As discussed in Section F.2, the packet size is important in getting full benefit of the network bandwidth. One might ask, "What is the natural size of messages?" Figure F.32(a) shows the size of messages for a commercial fluid dynamics simulation application, called Fluent, collected on an InfiniBand network at The Ohio State University's Network-Based Computer Laboratory. One plot is cumulative in messages sent and the other is cumulative in data bytes sent. Messages in this graph are message passing interface (MPI) units of information, which gets divided into InfiniBand maximum transfer units (packets) transferred over the network. As shown, the maximum message size is over 512 KB, but approximately 90% of the messages are less than 512 bytes. Messages of 2 KB represent approximately 50% of the bytes transferred. An Integer Sort application kernel in the NAS Parallel

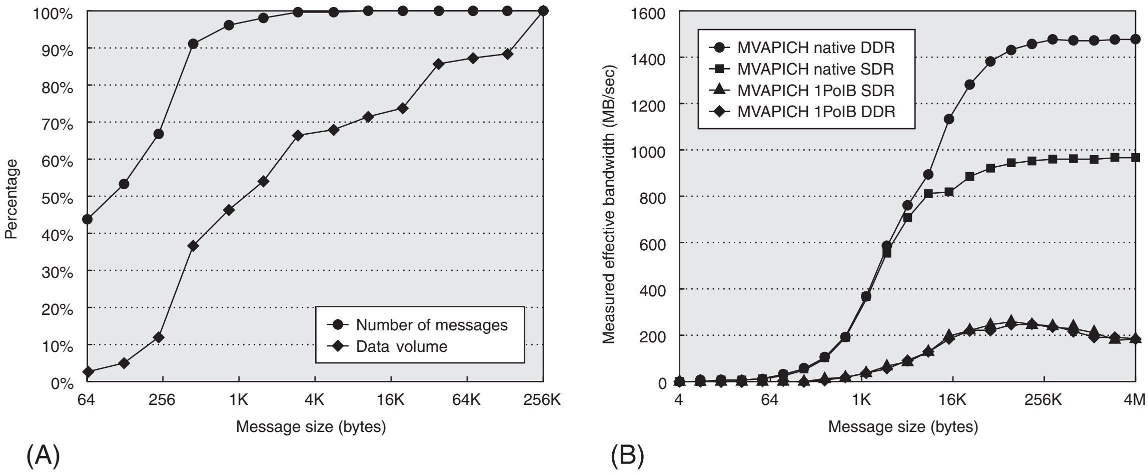


Figure F.32 Data collected by D.K. Panda, S. Sur, and L. Chai (2005) in the Network-Based Computing Laboratory at The Ohio State University. (a) Cumulative percentage of messages and volume of data transferred as message size varies for the Fluent application (*www.fluent.com*). Each *x*-axis entry includes all bytes up to the next one; for example, 128 represents 1 byte to 128 bytes. About 90% of the messages are less than 512 bytes, which represents about 40% of the total bytes transferred. (b) Effective bandwidth versus message size measured on SDR and DDR InfiniBand networks running MVAPICH (*http://nowlab.cse.ohio-state.edu/projects/mpi-iba*) with OS bypass (native) and without (IPoIB).

Benchmark suite is also measured to have about 75% of its messages below 512 bytes (plots not shown). Many applications send far more small messages than large ones, particularly since requests and acknowledgments are more frequent than data responses and block writes.

InfiniBand reduces protocol processing overhead by allowing it to be offloaded from the host computer to a controller on the InfiniBand network interface card. The benefits of protocol offloading and bypassing the operating system are shown in Figure F.32(b) for MVAPICH, a widely used implementation of MPI over InfiniBand. Effective bandwidth is plotted against message size for MVAPICH configured in two modes and two network speeds. One mode runs IPoIB, in which InfiniBand communication is handled by the IP layer implemented by the host's operating system (i.e., no OS bypass). The other mode runs MVAPICH directly over VAPI, which is the native Mellanox InfiniBand interface that offloads transport protocol processing to the channel adapter hardware (i.e., OS bypass). Results are shown for 10 Gbps single data rate (SDR) and 20 Gbps double data rate (DDR) InfiniBand networks. The results clearly show that offloading the protocol processing and bypassing the OS significantly reduce sending and receiving overhead to allow near wire-speed effective bandwidth to be achieved.

Ethernet: *The* Local Area Network

Ethernet has been extraordinarily successful as a LAN—from the 10 Mbit/sec standard proposed in 1978 used practically everywhere today to the more recent 10 Gbit/sec standard that will likely be widely used. Many classes of computers include Ethernet as a standard communication interface. Ethernet, codified as IEEE standard 802.3, is a packet-switched network that routes packets using the destination address. It was originally designed for coaxial cable but today uses primarily Cat5E copper wire, with optical fiber reserved for longer distances and higher bandwidths. There is even a wireless version (802.11), which is testimony to its ubiquity.

Over a 20-year span, computers became thousands of times faster than they were in 1978, but the shared media Ethernet network remained the same. Hence, engineers had to invent temporary solutions until a faster, higher-bandwidth network became available. One solution was to use multiple Ethernets to interconnect machines and to connect those Ethernets with internetworking devices that could transfer traffic from one Ethernet to another, as needed. Such devices allow individual Ethernets to operate in parallel, thereby increasing the aggregate interconnection bandwidth of a collection of computers. In effect, these devices provide similar functionality to the switches described previously for point-to-point networks.

Figure F.33 shows the potential parallelism that can be gained. Depending on how they pass traffic and what kinds of interconnections they can join together, these devices have different names:

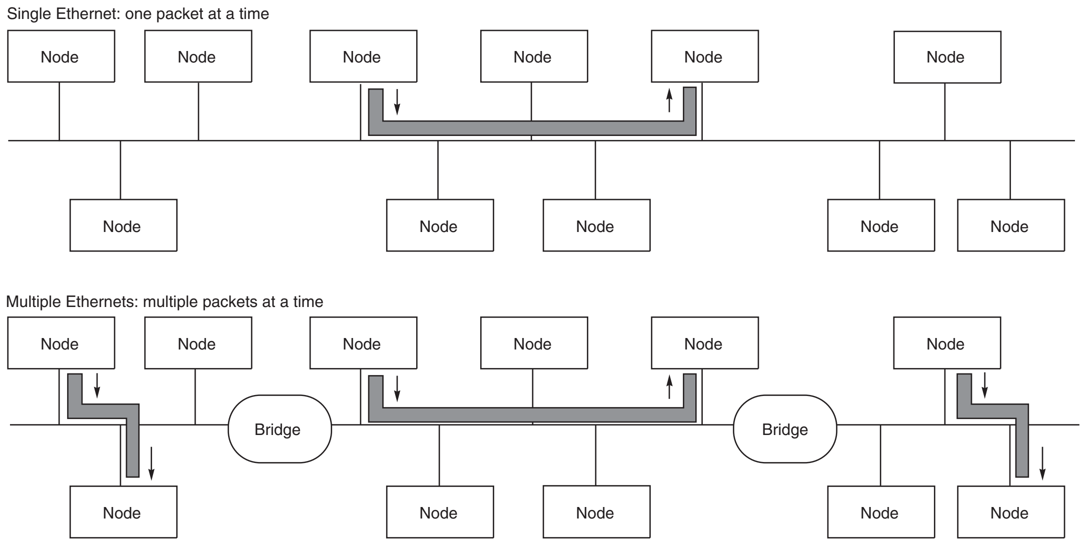

Single Ethernet: one packet at a time
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
Node
Multiple Ethernets: multiple packets at a time
Node
Node
Node
Node
Node
Node
Bridge
Bridge
Node
Node
Node
Node
Node

Figure F.33 The potential increased bandwidth of using many Ethernets and bridges.

- *Bridges*—These devices connect LANs together, passing traffic from one side to another depending on the addresses in the packet. Bridges operate at the Ethernet protocol level and are usually simpler and cheaper than routers, discussed next. Using the notation of the OSI model described in the next section (see Figure F.36 on page F-85), bridges operate at layer 2, the data link layer.
- Routers or *gateways*—These devices connect LANs to WANs, or WANs to WANs, and resolve incompatible addressing. Generally slower than bridges, they operate at OSI layer 3, the network layer. WAN routers divide the network into separate smaller subnets, which simplifies manageability and improves security.

The final internetworking devices are *hubs*, but they merely extend multiple segments into a single LAN. Thus, hubs do not help with performance, as only one message can transmit at a time. Hubs operate at OSI layer 1, called the physical

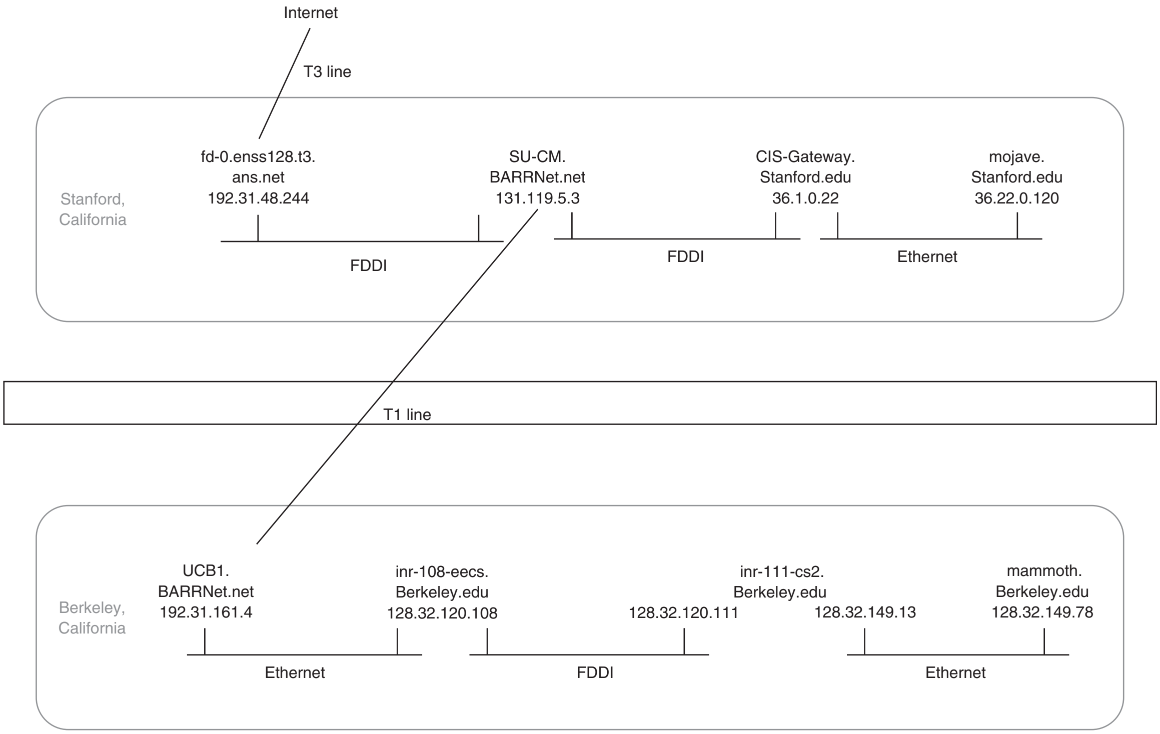

Internet
T3 line
Stanford, California
fd-0.enss128.t3.
ans.net
192.31.48.244
SU-CM.
BARRNet.net
131.119.5.3
CIS-Gateway.
Stanford.edu
36.1.0.22
mojave.
Stanford.edu
36.22.0.120
FDDI
FDDI
Ethernet
T1 line
Berkeley, California
UCB1.
BARRNet.net
192.31.161.4
inr-108-eecs.
Berkeley.edu
128.32.120.108
inr-111-cs2.
Berkeley.edu
128.32.120.111
128.32.149.13
mammoth.
Berkeley.edu
128.32.149.78
Ethernet
FDDI
Ethernet

Figure F.34 The connection established between mojave.stanford.edu and mammoth.berkeley.edu (1995). FDDI is a 100 Mbit/sec LAN, while a T1 line is a 1.5 Mbit/sec telecommunications line and a T3 is a 45 Mbit/sec telecommunications line. BARRNet stands for Bay Area Research Network. Note that inr-111-cs2.Berkeley.edu is a router with two Internet addresses, one for each port.

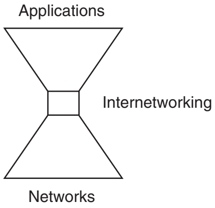


Figure F.35 The role of internetworking. The width indicates the relative number of items at each level.

Layer number	Layer name	Main function	Example protocol	Network component
7	Application	Used for applications specifically written to run over the network	FTP, DNS, NFS, http	Gateway, smart switch
6	Presentation	Translates from application to network format, and *vice versa*		Gateway
5	Session	Establishes, maintains, and ends sessions across the network	Named pipes, RPC	Gateway
4	Transport	Additional connection below the session layer	TCP	Gateway
3	Network	Translates logical network address and names to their physical address (e.g., computer name to MAC address)	IP	Router, ATM switch
2	Data Link	Turns packets into raw bits and at the receiving end turns bits into packets	Ethernet	Bridge, network interface card
1	Physical	Transmits raw bit stream over physical cable	IEEE 802	Hub

Figure F.36 The OSI model layers. Based on *www.geocities.com/SiliconValley/Monitor/3131/ne/osimodel.html*.

layer. Since these devices were not planned as part of the Ethernet standard, their ad hoc nature has added to the difficulty and cost of maintaining LANs.

As of 2011, Ethernet link speeds are available at 10, 100, 10,000, and 100,000 Mbits/sec. Although 10 and 100 Mbits/sec Ethernets share the media with multiple devices, 1000 Mbits/sec and above Ethernets rely on point-to-point links and switches. Ethernet switches normally use some form of store-and-forward.

Ethernet has no real flow control, dating back to its first instantiation. It originally used carrier sensing with exponential back-off (see page F-23) to arbitrate for the shared media. Some switches try to use that interface to retrofit their version of flow control, but flow control is not part of the Ethernet standard.

Wide Area Network: ATM

Asynchronous Transfer Mode (ATM) is a wide area networking standard set by the telecommunications industry. Although it flirted as competition to Ethernet as a LAN in the 1990s, ATM has since retreated to its WAN stronghold.

The telecommunications standard has scalable bandwidth built in. It starts at 155 Mbits/sec and scales by factors of 4 to 620 Mbits/sec, 2480 Mbits/sec, and so on. Since it is a WAN, ATM's medium is fiber, both single mode and multimode. Although it is a switched medium, unlike the other examples it relies on virtual connections for communication. ATM uses virtual channels for routing to multiplex different connections on a single network segment, thereby avoiding the inefficiencies of conventional connection-based networking. The WAN focus also led to store-and-forward switching. Unlike the other protocols, Figure F.30 shows ATM has a small, fixed-sized packet with 48 bytes of payload. It uses a credit-based flow control scheme as opposed to IP routers that do not implement flow control.

The reason for virtual connections and small packets is quality of service. Since the telecommunications industry is concerned about voice traffic, predictability matters as well as bandwidth. Establishing a virtual connection has less variability than connectionless networking, and it simplifies store-and-forward switching. The small, fixed packet also makes it simpler to have fast routers and switches. Toward that goal, ATM even offers its own protocol stack to compete with TCP/IP. Surprisingly, even though the switches are simple, the ATM suite of protocols is large and complex. The dream was a seamless infrastructure from LAN to WAN, avoiding the hodgepodge of routers common today. That dream has faded from inspiration to nostalgia.

F.9 Internetworking

Undoubtedly one of the most important innovations in the communications community has been internetworking. It allows computers on independent and incompatible networks to communicate reliably and efficiently. Figure F.34 illustrates the need to traverse between networks. It shows the networks and machines involved in transferring a file from Stanford University to the University of California at Berkeley, a distance of about 75 km.

The low cost of internetworking is remarkable. For example, it is vastly less expensive to send electronic mail than to make a coast-to-coast telephone call and leave a message on an answering machine. This dramatic cost improvement is achieved using the same long-haul communication lines as the telephone call, which makes the improvement even more impressive.

The enabling technologies for internetworking are software standards that allow reliable communication without demanding reliable networks. The underlying principle of these successful standards is that they were composed as a hierarchy of layers, each layer taking responsibility for a portion of the overall communication task. Each computer, network, and switch implements its layer of the standards, relying on the other components to faithfully fulfill their responsibilities. These layered software standards are called protocol families or protocol suites. They enable applications to work with any interconnection without extra work by the application programmer. Figure F.35 suggests the hierarchical model of communication.

The most popular internetworking standard is TCP/IP (Transmission Control Protocol/Internet Protocol). This protocol family is the basis of the humbly named Internet, which connects hundreds of millions of computers around the world. This popularity means TCP/IP is used even when communicating locally across compatible networks; for example, the network file system (NFS) uses IP even though it is very likely to be communicating across a homogenous LAN such as Ethernet. We use TCP/IP as our protocol family example; other protocol families follow similar lines. Section F.13 gives the history of TCP/IP.

The goal of a family of protocols is to simplify the standard by dividing responsibilities hierarchically among layers, with each layer offering services needed by the layer above. The application program is at the top, and at the bottom is the physical communication medium, which sends the bits. Just as abstract data types simplify the programmer's task by shielding the programmer from details of the implementation of the data type, this layered strategy makes the standard easier to understand.

There were many efforts at network protocols, which led to confusion in terms. Hence, Open Systems Interconnect (OSI) developed a model that popularized describing networks as a series of layers. Figure F.36 shows the model. Although all protocols do not exactly follow this layering, the nomenclature for the different layers is widely used. Thus, you can hear discussions about a simple layer 3 switch versus a layer 7 smart switch.

The key to protocol families is that communication occurs logically at the same level of the protocol in both sender and receiver, but services of the lower level implement it. This style of communication is called *peer-to-peer*. As an analogy, imagine that General A needs to send a message to General B on the battlefield. General A writes the message, puts it in an envelope addressed to General B, and gives it to a colonel with orders to deliver it. This colonel puts it in an envelope, and writes the name of the corresponding colonel who reports to General B, and gives it to a major with instructions for delivery. The major does the same thing and gives it to a captain, who gives it to a lieutenant, who gives it to a sergeant. The sergeant takes the envelope from the lieutenant, puts it into an envelope with the name of a sergeant who is in General B's division, and finds a private with orders to take the large envelope. The private borrows a motorcycle and delivers the envelope to the other sergeant. Once it arrives, it is passed up the chain of command, with each person removing an outer envelope with his name on it and passing on the inner envelope to his superior. As far as General B can tell, the note is from another general. Neither general knows who was involved in transmitting the envelope, nor how it was transported from one division to the other.

Protocol families follow this analogy more closely than you might think, as Figure F.37 shows. The original message includes a header and possibly a trailer sent by the lower-level protocol. The next-lower protocol in turn adds its own header to the message, possibly breaking it up into smaller messages if it is too large for this layer. Reusing our analogy, a long message from the general is divided and placed in several envelopes if it could not fit in one. This division of the message and appending of headers and trailers continues until the message

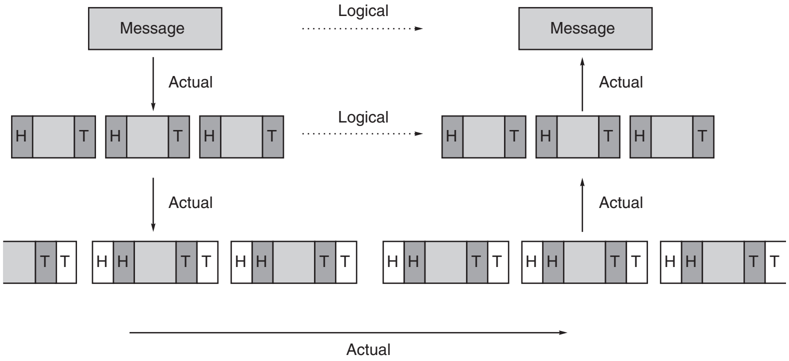


Figure F.37 A generic protocol stack with two layers. Note that communication is peer-to-peer, with headers and trailers for the peer added at each sending layer and removed by each receiving layer. Each layer offers services to the one above to shield it from unnecessary details.

descends to the physical transmission medium. The message is then sent to the destination. Each level of the protocol family on the receiving end will check the message at its level and peel off its headers and trailers, passing it on to the next higher level and putting the pieces back together. This nesting of protocol layers for a specific message is called a *protocol stack*, reflecting the last in, first out nature of the addition and removal of headers and trailers.

As in our analogy, the danger in this layered approach is the considerable latency added to message delivery. Clearly, one way to reduce latency is to reduce the number of layers, but keep in mind that protocol families define a standard but do not force how to implement the standard. Just as there are many ways to implement an instruction set architecture, there are many ways to implement a protocol family.

Our protocol stack example is TCP/IP. Let's assume that the bottom protocol layer is Ethernet. The next level up is the Internet Protocol or IP layer; the official term for an IP packet is a datagram. The IP layer routes the datagram to the destination machine, which may involve many intermediate machines or switches. IP makes a best effort to deliver the packets but does not guarantee delivery, content, or order of datagrams. The TCP layer above IP makes the guarantee of reliable, in-order delivery and prevents corruption of datagrams.

Following the example in Figure F.37, assume an application program wants to send a message to a machine via an Ethernet. It starts with TCP. The largest number of bytes that can be sent at once is 64 KB. Since the data may be much larger than 64 KB, TCP must divide them into smaller segments and reassemble them in proper order upon arrival. TCP adds a 20-byte header (Figure F.38) to every datagram and passes them down to IP. The IP layer above the physical layer adds a 20-byte header, also shown in Figure F.38. The data sent down from the IP level

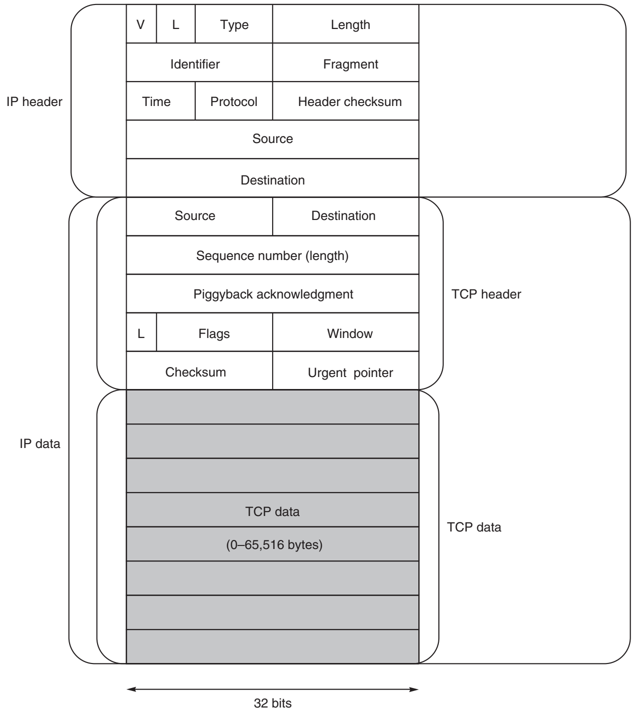


Figure F.38 The headers for IP and TCP. This drawing is 32 bits wide. The standard headers for both are 20 bytes, but both allow the headers to optionally lengthen for rarely transmitted information. Both headers have a length of header field (L) to accommodate the optional fields, as well as source and destination fields. The length field of the whole datagram is in a separate length field in IP, while TCP combines the length of the datagram with the sequence number of the datagram by giving the sequence number in bytes. TCP uses the checksum field to be sure that the datagram is not corrupted, and the sequence number field to be sure the datagrams are assembled into the proper order when they arrive. IP provides checksum error detection only for the header, since TCP has protected the rest of the packet. One optimization is that TCP can send a sequence of datagrams before waiting for permission to send more. The number of datagrams that can be sent without waiting for approval is called the *window*, and the window field tells how many bytes may be sent beyond the byte being acknowledged by this datagram. TCP will adjust the size of the window depending on the success of the IP layer in sending datagrams; the more reliable and faster it is, the larger TCP makes the window. Since the window slides forward as the data arrive and are acknowledged, this technique is called a *sliding window protocol*. The piggyback acknowledgment field of TCP is another optimization. Since some applications send data back and forth over the same connection, it seems wasteful to send a datagram containing only an acknowledgment. This piggyback field allows a datagram carrying data to also carry the acknowledgment for a previous transmission, "piggybacking" on top of a data transmission. The urgent pointer field of TCP gives the address within the datagram of an important byte, such as a break character. This pointer allows the application software to skip over data so that the user doesn't have to wait for all prior data to be processed before seeing a character that tells the software to stop. The identifier field and fragment field of IP allow intermediary machines to break the original datagram into many smaller datagrams. A unique identifier is associated with the original datagram and placed in every fragment, with the fragment field saying which piece is which. The time-to-live field allows a datagram to be killed off after going through a maximum number of intermediate switches no matter where it is in the network. Knowing the maximum number of hops that it will take for a datagram to arrive—if it ever arrives—simplifies the protocol software. The protocol field identifies which possible upper layer protocol sent the IP datagram; in our case, it is TCP. The V (for version) and type fields allow different versions of the IP protocol software for the network. Explicit version numbering is included so that software can be upgraded gracefully machine by machine, without shutting down the entire network. Nowadays, version six of the Internet protocol (IPv6) was widely used.

to the Ethernet are sent in packets with the format shown in Figure F.30. Note that the TCP packet appears inside the data portion of the IP datagram, just as Figure F.37 suggests.

F.10 Crosscutting Issues for Interconnection Networks

This section describes five topics discussed in other chapters that are fundamentally impacted by interconnection networks, and *vice versa*.

Density-Optimized Processors versus SPEC-Optimized Processors

Given that people all over the world are accessing Web sites, it doesn't really matter where servers are located. Hence, many servers are kept at *collocation sites*, which charge by network bandwidth reserved and used and by space occupied and power consumed. Desktop microprocessors in the past have been designed to be as fast as possible at whatever heat could be dissipated, with little regard for the size of the package and surrounding chips. In fact, some desktop microprocessors from Intel and AMD as recently as 2006 burned as much as 130 watts! Floor space efficiency was also largely ignored. As a result of these priorities, power is a major cost for collocation sites, and processor density is limited by the power consumed and dissipated, including within the interconnect!

With the proliferation of portable computers (notebook sales exceeded desktop sales for the first time in 2005) and their reduced power consumption and cooling demands, the opportunity exists for using this technology to create considerably denser computation. For instance, the power consumption for the Intel Pentium M in 2006 was 25 watts, yet it delivered performance close to that of a desktop microprocessor for a wide set of applications. It is therefore conceivable that performance per watt or performance per cubic foot could replace performance per microprocessor as the important figure of merit. The key is that many applications already make use of large clusters, so it is possible that replacing 64 power-hungry processors with, say, 256 power-efficient processors could be cheaper yet be software compatible. This places greater importance on power- and performance-efficient interconnection network design.

The Google cluster is a prime example of this migration to many "cooler" processors versus fewer "hotter" processors. It uses racks of up to 80 Intel Pentium III 1 GHz processors instead of more power-hungry high-end processors. Other examples include blade servers consisting of 1-inch-wide by 7-inch-high rack unit blades designed based on mobile processors. The HP ProLiant BL10e G2 blade server supports up to 20 1-GHz ultra-low-voltage Intel Pentium M processors with a 400-MHz front-side bus, 1-MB L2 cache, and up to 1 GB memory. The Fujitsu Primergy BX300 blade server supports up to 20 1.4- or 1.6-GHz Intel Pentium M processors, each with 512 MB of memory expandable to 4 GB.

Smart Switches versus Smart Interface Cards

Figure F.39 shows a trade-off as to where intelligence can be located within a network. Generally, the question is whether to have either smarter network interfaces or smarter switches. Making one smarter generally makes the other simpler and less expensive. By having an inexpensive interface, it was possible for Ethernet to become standard as part of most desktop and server computers. Lower-cost switches were made available for people with small configurations, not needing sophisticated forwarding tables and spanning-tree protocols of larger Ethernet switches.

Myrinet followed the opposite approach. Its switches are dumb components that, other than implementing flow control and arbitration, simply extract the first byte from the packet header and use it to directly select the output port. No routing tables are implemented, so the intelligence is in the network interface cards (NICs). The NICs are responsible for providing support for efficient communication and for implementing a distributed protocol for network (re)configuration. InfiniBand takes a hybrid approach by offering lower-cost, less sophisticated interface cards called target channel adapters (or TCAs) for less demanding devices such as disks—in the hope that it can be included within some I/O devices—and by offering more expensive, powerful interface cards for hosts called host channel adapters (or HCAs). The switches implement routing tables.

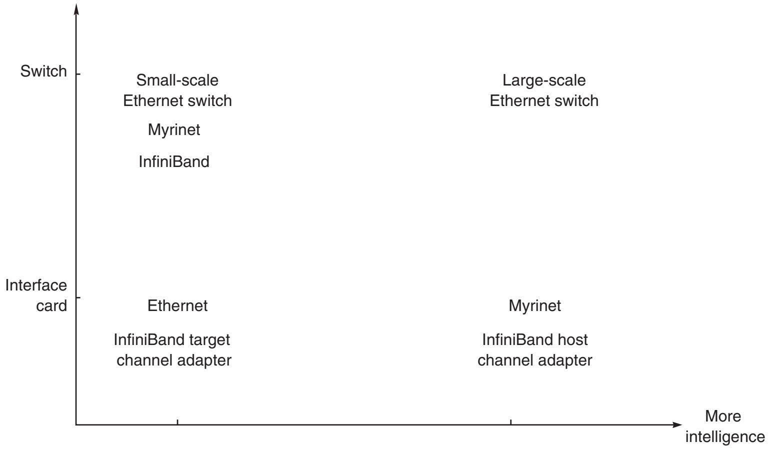

Switch
Small-scale Ethernet switch
Large-scale Ethernet switch
Myrinet
InfiniBand
Interface card
Ethernet
Myrinet
InfiniBand target channel adapter
InfiniBand host channel adapter
More intelligence

Figure F.39 Intelligence in a network: switch versus network interface card. Note that Ethernet switches come in two styles, depending on the size of the network, and that InfiniBand network interfaces come in two styles, depending on whether they are attached to a computer or to a storage device. Myrinet is a proprietary system area network.

Protection and User Access to the Network

A challenge is to ensure safe communication across a network without invoking the operating system in the common case. The Cray Research T3D supercomputer offers an interesting case study. Like the more recent Cray X1E, the T3D supports a global address space, so loads and stores can access memory across the network. Protection is ensured because each access is checked by the TLB. To support transfer of larger objects, a block transfer engine (BLT) was added to the hardware. Protection of access requires invoking the operating system before using the BLT to check the range of accesses to be sure there will be no protection violations.

Figure F.40 compares the bandwidth delivered as the size of the object varies for reads and writes. For very large reads (e.g., 512 KB), the BLT achieves the highest performance: 140 MB/sec. But simple loads get higher performance for 8 KB or less. For the write case, both achieve a peak of 90 MB/sec, presumably because of the limitations of the memory bus. But, for writes, the BLT can only match the performance of simple stores for transfers of 2 MB; anything smaller and it's faster to send stores. Clearly, a BLT that can avoid invoking the operating system in the common case would be more useful.

Efficient Interface to the Memory Hierarchy versus the Network

Traditional evaluations of processor performance, such as SPECint and SPECfp, encourage integration of the memory hierarchy with the processor as the efficiency of the memory hierarchy translates directly into processor performance. Hence,

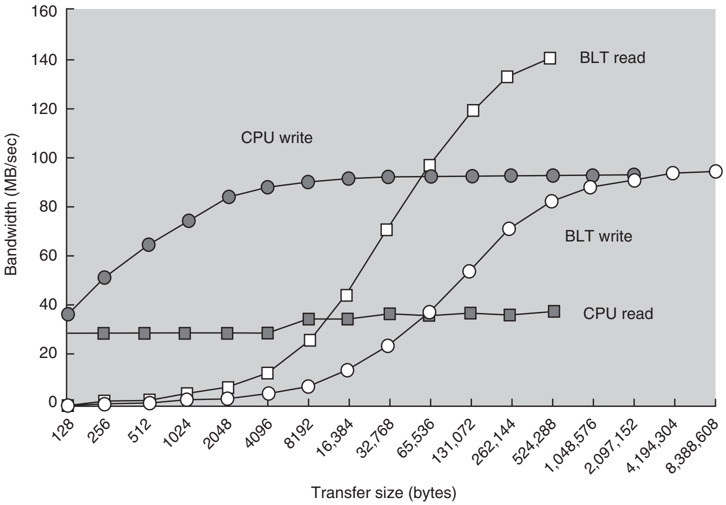


Figure F.40 Bandwidth versus transfer size for simple memory access instructions versus a block transfer device on the Cray Research T3D. (From Arpaci et al. [1995].)

microprocessors have multiple levels of caches on chip along with buffers for writes. Because benchmarks such as SPECint and SPECfp do not reward good interfaces to interconnection networks, many machines make the access time to the network delayed by the full memory hierarchy. Writes must lumber their way through full write buffers, and reads must go through the cycles of first-, second-, and often third-level cache misses before reaching the interconnection network. This hierarchy results in newer systems having higher latencies to the interconnect than older machines.

Let's compare three machines from the past: a 40-MHz SPARCstation-2, a 50-MHz SPARCstation-20 without an external cache, and a 50-MHz SPARCstation-20 with an external cache. According to SPECint95, this list is in order of increasing performance. The time to access the I/O bus (S-bus), however, increases in this sequence: 200 ns, 500 ns, and 1000 ns. The SPARCstation-2 is fastest because it has a single bus for memory and I/O, and there is only one level to the cache. The SPARCstation-20 memory access must first go over the memory bus (M-bus) and then to the I/O bus, adding 300 ns. Machines with a second-level cache pay an extra penalty of 500 ns before accessing the I/O bus.

Compute-Optimized Processors versus Receiver Overhead

The overhead to receive a message likely involves an interrupt, which bears the cost of flushing and then restarting the processor pipeline, if not offloaded. As mentioned earlier, reading network status and receiving data from the network interface likely operate at cache miss speeds. If microprocessors become more superscalar and go to even faster clock rates, the number of missed instruction issue opportunities per message reception will likely rise to unacceptable levels.

F.11 Fallacies and Pitfalls

Myths and hazards are widespread with interconnection networks. This section mentions several warnings, so proceed carefully.

Fallacy *The interconnection network is very fast and does not need to be improved*

The interconnection network provides certain functionality to the system, very much like the memory and I/O subsystems. It should be designed to allow processors to execute instructions at the maximum rate. The interconnection network subsystem should provide high enough bandwidth to keep from continuously entering saturation and becoming an overall system bottleneck.

In the 1980s, when wormhole switching was introduced, it became feasible to design large-diameter topologies with single-chip switches so that the bandwidth capacity of the network was not the limiting factor. This led to the flawed belief that interconnection networks need no further improvement.

Since the 1980s, much attention has been placed on improving processor performance, but comparatively less has been focused on interconnection networks. As technology advances, the interconnection network tends to represent an increasing fraction of system resources, cost, power consumption, and various other attributes that impact functionality and performance. Scaling the bandwidth simply by overdimensioning certain network parameters is no longer a cost-viable option. Designers must carefully consider the end-to-end interconnection network design in concert with the processor, memory, and I/O subsystems in order to achieve the required cost, power, functionality, and performance objectives of the entire system. An obvious case in point is multicore processors with on-chip networks.

Fallacy *Bisection bandwidth is an accurate cost constraint of a network*

Despite being very popular, bisection bandwidth has never been a practical constraint on the implementation of an interconnection network, although it may be one in future designs. It is more useful as a performance measure than as a cost measure. Chip pin-outs are the more realistic bandwidth constraint.

Pitfall *Using bandwidth (in particular, bisection bandwidth) as the only measure of network performance*

It seldom is the case that aggregate network bandwidth (likewise, network bisection bandwidth) is the end-to-end bottlenecking point across the network. Even if it were the case, networks are almost never 100% efficient in transporting packets across the bisection (i.e., $\rho < 100\%$) nor at receiving them at network endpoints (i.e., $\sigma < 100\%$). The former is highly dependent upon routing, switching, arbitration, and other such factors while both the former and the latter are highly dependent upon traffic characteristics. Ignoring these important factors and concentrating only on raw bandwidth can give very misleading performance predictions. For example, it is perfectly conceivable that a network could have higher aggregate bandwidth and/or bisection bandwidth relative to another network but also have lower measured performance!

Apparently, given sophisticated protocols like TCP/IP that maximize delivered bandwidth, many network companies believe that there is only one figure of merit for networks. This may be true for some applications, such as video streaming, where there is little interaction between the sender and the receiver. Many applications, however, are of a request-response nature, and so for every large message there must be one or more small messages. One example is NFS.

Figure F.41 compares a shared 10-Mbit/sec Ethernet LAN to a switched 155-Mbit/sec ATM LAN for NFS traffic. Ethernet drivers were better tuned than the ATM drivers, such that 10-Mbit/sec Ethernet was faster than 155-Mbit/sec ATM for payloads of 512 bytes or less. Figure F.41 shows the overhead time, transmission time, and total time to send all the NFS messages over Ethernet and ATM. The peak link speed of ATM is 15 times faster, and the measured link speed for 8-KB messages is almost 9 times faster. Yet, the higher overheads offset the benefits so that ATM would transmit NFS traffic only 1.2 times faster.

		Overhead (sec)			Transmission (sec)		Total time (sec)	
Size	Number of messages	ATM	Ethernet	Number of data bytes	ATM	Ethernet	ATM	Ethernet
32	771,060	532	389	33,817,052	4	48	536	436
64	56,923	39	29	4,101,088	0	5	40	34
96	4,082,014	2817	2057	428,346,316	46	475	2863	2532
128	5,574,092	3846	2809	779,600,736	83	822	3929	3631
160	328,439	227	166	54,860,484	6	56	232	222
192	16,313	11	8	3,316,416	0	3	12	12
224	4820	3	2	1,135,380	0	1	3	4
256	24,766	17	12	9,150,720	1	9	18	21
512	32,159	22	16	25,494,920	3	23	25	40
1024	69,834	48	35	70,578,564	8	72	56	108
1536	8842	6	4	15,762,180	2	14	8	19
2048	9170	6	5	20,621,760	2	19	8	23
2560	20,206	14	10	56,319,740	6	51	20	61
3072	13,549	9	7	43,184,992	4	39	14	46
3584	4200	3	2	16,152,228	2	14	5	17
4096	67,808	47	34	285,606,596	29	255	76	290
5120	6143	4	3	35,434,680	4	32	8	35
6144	5858	4	3	37,934,684	4	34	8	37
7168	4140	3	2	31,769,300	3	28	6	30
8192	287,577	198	145	2,390,688,480	245	2132	444	2277
Total	11,387,913	7858	5740	4,352,876,316	452	4132	8310	9872

Figure F.41 Total time on a 10-Mbit Ethernet and a 155-Mbit ATM, calculating the total overhead and transmission time separately. Note that the size of the headers needs to be added to the data bytes to calculate transmission time. The higher overhead of the software driver for ATM offsets the higher bandwidth of the network. These measurements were performed in 1994 using SPARCstation 10s, the ForeSystems SBA-200 ATM interface card, and the Fore Systems ASX-200 switch. (NFS measurements taken by Mike Dahlin of the University of California–Berkeley.)

Pitfall *Not providing sufficient reception link bandwidth, which causes the network end nodes to become even more of a bottleneck to performance*

Unless the traffic pattern is a permutation, several packets will concurrently arrive at some destinations when most source devices inject traffic, thus producing contention. If this problem is not addressed, contention may turn into congestion that will spread across the network. This can be dealt with by analyzing traffic patterns and providing extra reception bandwidth. For example, it is possible to implement more reception bandwidth than injection bandwidth. The IBM Blue Gene/L, for example, implements an on-chip switch with 7-bit injection and 12-bit reception links, where the reception BW equals the aggregate switch input link BW.

Pitfall *Using high-performance network interface cards but forgetting about the I/O subsystem that sits between the network interface and the host processor*

This issue is related to the previous one. Messages are usually composed in user space buffers and later sent by calling a send function from the communications library. Alternatively, a cache controller implementing a cache coherence protocol may compose a message in some SANs and in OCNs. In both cases, messages have to be copied to the network interface memory before transmission. If the I/O bandwidth is lower than the link bandwidth or introduces significant overhead, this is going to affect communication performance significantly. As an example, the first 10-Gigabit Ethernet cards in the market had a PCI-X bus interface for the system with a significantly lower bandwidth than 10 Gbps.

Fallacy *Zero-copy protocols do not require copying messages or fragments from one buffer to another*

Traditional communication protocols for computer networks allow access to communication devices only through system calls in supervisor mode. As a consequence of this, communication routines need to copy the corresponding message from the user buffer to a kernel buffer when sending a message. Note that the communication protocol may need to keep a copy of the message for retransmission in case of error, and the application may modify the contents of the user buffer once the system call returns control to the application. This buffer-to-buffer copy is eliminated in zero-copy protocols because the communication routines are executed in user space and protocols are much simpler.

However, messages still need to be copied from the application buffer to the memory in the network interface card (NIC) so that the card hardware can transmit it from there through to the network. Although it is feasible to eliminate this copy by allocating application message buffers directly in the NIC memory (and, indeed, this is done in some protocols), this may not be convenient in current systems because access to the NIC memory is usually performed through the I/O subsystem, which usually is much slower than accessing main memory. Thus, it is generally more efficient to compose the message in main memory and let DMA devices take care of the transfer to the NIC memory.

Moreover, what few people count is the copy from where the message fragments are computed (usually the ALU, with results stored in some processor register) to main memory. Some systolic-like architectures in the 1980s, like the iWarp, were able to directly transmit message fragments from the processor to the network, effectively eliminating all the message copies. This is the approach taken in the Cray X1E shared-memory multiprocessor supercomputer.

Similar comments can be made regarding the reception side; however, this does not mean that zero-copy protocols are inefficient. These protocols represent the most efficient kind of implementation used in current systems.

Pitfall *Ignoring software overhead when determining performance*

Low software overhead requires cooperation with the operating system as well as with the communication libraries, but even with protocol offloading it continues to dominate the hardware overhead and must not be ignored. Figures F.32 and F.41 give two examples, one for a SAN standard and the other for a WAN standard. Other examples come from proprietary SANs for supercomputers. The Connection Machine CM-5 supercomputer in the early 1990s had a software overhead of 20 μs to send a message and a hardware overhead of only 0.5 μs. The first Intel Paragon supercomputer built in the early 1990s had a hardware overhead of just 0.2 μs, but the initial release of the software had an overhead of 250 μs. Later releases reduced this overhead down to 25 μs and, more recently, down to only a few microseconds, but this still dominates the hardware overhead. The IBM Blue Gene/L has an MPI sending/receiving overhead of approximately 3 μs, only a third of which (at most) is attributed to the hardware.

This pitfall is simply Amdahl's law applied to networks: Faster network hardware is superfluous if there is not a corresponding decrease in software overhead. The software overhead is much reduced these days with OS bypass, lightweight protocols, and protocol offloading down to a few microseconds or less, typically, but it remains a significant factor in determining performance.

Fallacy *MINs are more cost-effective than direct networks*

A MIN is usually implemented using significantly fewer switches than the number of devices that need to be connected. On the other hand, direct networks usually include a switch as an integral part of each node, thus requiring as many switches as nodes to interconnect. However, nothing prevents the implementation of nodes with multiple computing devices on it (e.g., a multicore processor with an on-chip switch) or with several devices attached to each switch (i.e., bristling). In these cases, a direct network may be as (or even more) cost-effective as a MIN. Note that, for a MIN, several network interfaces may be required at each node to match the bandwidth delivered by the multiple links per node provided by the direct network.

Fallacy *Low-dimensional direct networks achieve higher performance than high-dimensional networks such as hypercubes*

This conclusion was drawn by several studies that analyzed the optimal number of dimensions under the main physical constraint of bisection bandwidth. However, most of those studies did not consider link pipelining, considered only very short links, and/or did not consider switch architecture design constraints. The misplaced assumption that bisection bandwidth serves as the main limit did not help matters. Nowadays, most researchers and designers believe that high-radix switches are more cost-effective than low-radix switches, including some who concluded the opposite before.

Fallacy *Wormhole switching achieves better performance than other switching techniques*

Wormhole switching delivers the same no-load latency as other pipelined switching techniques, like virtual cut-through switching. The introduction of wormhole switches in the late 1980s coinciding with a dramatic increase in network bandwidth led many to believe that wormhole switching was the main reason for the performance boost. Instead, most of the performance increase came from a drastic increase in link bandwidth, which, in turn, was enabled by the ability of wormhole switching to buffer packet fragments using on-chip buffers, instead of using the node's main memory or some other off-chip source for that task. More recently, much larger on-chip buffers have become feasible, and virtual cutthrough achieved the same no-load latency as wormhole while delivering much higher throughput. This did not mean that wormhole switching was dead. It continues to be the switching technique of choice for applications in which only small buffers should be used (e.g., perhaps for on-chip networks).

Fallacy *Implementing a few virtual channels always increases throughput by allowing packets to pass through blocked packets ahead*

In general, implementing a few virtual channels in a wormhole switch is a good idea because packets are likely to pass blocked packets ahead of them, thus reducing latency and significantly increasing throughput. However, the improvements are not as dramatic for virtual cut-through switches. In virtual cut-through, buffers should be large enough to store several packets. As a consequence, each virtual channel may introduce HOL blocking, possibly degrading performance at high loads. Adding virtual channels increases cost, but it may deliver little additional performance unless there are as many virtual channels as switch ports and packets are mapped to virtual channels according to their destination (i.e., virtual output queueing). It is certainly the case that virtual channels can be useful in virtual cut-through networks to segregate different traffic classes, which can be very beneficial. However, multiplexing the packets over a physical link on a flit-by-flit basis causes all the packets from different virtual channels to get delayed. The average packet delay is significantly shorter if multiplexing takes place on a packet-by-packet basis, but in this case packet size should be bounded to prevent any one packet from monopolizing the majority of link bandwidth.

Fallacy *Adaptive routing causes out-of-order packet delivery, thus introducing too much overhead needed to reorder packets at the destination device*

Adaptive routing allows packets to follow alternative paths through the network depending on network traffic; therefore, adaptive routing usually introduces outof-order packet delivery. However, this does not necessarily imply that reordering packets at the destination device is going to introduce a large overhead, making adaptive routing not useful. For example, the most efficient adaptive routing algorithms to date support fully adaptive routing in some virtual channels but required deterministic routing to be implemented in some other virtual channels in order to prevent deadlocks (à la the IBM Blue Gene/L). In this case, it is very easy to select between adaptive and deterministic routing for each individual packet. A single bit in the packet header can indicate to the switches whether all the virtual channels can be used or only those implementing deterministic routing. This hardware support can be used as indicated below to eliminate packet reordering overhead at the destination.

Most communication protocols for parallel computers and clusters implement two different protocols depending on message size. For short messages, an eager protocol is used in which messages are directly transmitted, and the receiving nodes use some preallocated buffer to temporarily store the incoming message. On the other hand, for long messages, a rendezvous protocol is used. In this case, a control message is sent first, requesting the destination node to allocate a buffer large enough to store the entire message. The destination node confirms buffer allocation by returning an acknowledgment, and the sender can proceed with fragmenting the message into bounded-size packets, transmitting them to the destination.

If eager messages use only deterministic routing, it is obvious that they do not introduce any reordering overhead at the destination. On the other hand, packets belonging to a long message can be transmitted using adaptive routing. As every packet contains the sequence number within the message (or the offset from the beginning of the message), the destination node can store every incoming packet directly in its correct location within the message buffer, thus incurring no overhead with respect to using deterministic routing. The only thing that differs is the completion condition. Instead of checking that the last packet in the message has arrived, it is now necessary to count the arrived packets, notifying the end of reception when the count equals the message size. Taking into account that long messages, even if not frequent, usually consume most of the network bandwidth, it is clear that most packets can benefit from adaptive routing without introducing reordering overhead when using the protocol described above.

Fallacy *Adaptive routing by itself always improves network fault tolerance because it allows packets to follow alternative paths*

Adaptive routing by itself is not enough to tolerate link and/or switch failures. Some mechanism is required to detect failures and notify them, so that the routing logic could exclude faulty paths and use the remaining ones. Moreover, while a given link or switch failure affects a certain number of paths when using deterministic routing, many more source/destination pairs could be affected by the same failure when using adaptive routing. As a consequence of this, some switches implementing adaptive routing transition to deterministic routing in the presence of failures. In this case, failures are usually tolerated by sending messages through alternative paths from the source node. As an example, the Cray T3E implements direction-order routing to tolerate a few failures. This fault-tolerant routing technique avoids cycles in the use of resources by crossing directions in order (e.g., $X+$, $Y+$, $Z+$, $Z-$, $Y-$, then $X-$). At the same time, it provides an easy way to send packets through nonminimal paths, if necessary, to avoid crossing faulty components. For instance, a packet can be initially forwarded a few hops in the $X+$ direction even if it has to go in the $X-$ direction at some point later.

Pitfall *Trying to provide features only within the network versus end-to-end*

The concern is that of providing at a lower level the features that can only be accomplished at the highest level, thus only partially satisfying the communication demand. Saltzer, Reed, and Clark [1984] gave the end-to-end argument as follows:

> The function in question can completely and correctly be specified only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. [page 278]

Their example of the pitfall was a network at MIT that used several gateways, each of which added a checksum from one gateway to the next. The programmers of the application assumed that the checksum guaranteed accuracy, incorrectly believing that the message was protected while stored in the memory of each gateway. One gateway developed a transient failure that swapped one pair of bytes per million bytes transferred. Over time, the source code of one operating system was repeatedly passed through the gateway, thereby corrupting the code. The only solution was to correct infected source files by comparing them to paper listings and repairing code by hand! Had the checksums been calculated and checked by the application running on the end systems, safety would have been ensured.

There is a useful role for intermediate checks at the link level, however, provided that end-to-end checking is available. End-to-end checking may show that something is broken between two nodes, but it doesn't point to where the problem is. Intermediate checks can discover the broken component.

A second issue regards performance using intermediate checks. Although it is sufficient to retransmit the whole in case of failures from the end point, it can be much faster to retransmit a portion of the message at an intermediate point rather than wait for a time-out and a full message retransmit at the end point.

Pitfall *Relying on TCP/IP for all networks, regardless of latency, bandwidth, or software requirements*

The network designers on the first workstations decided it would be elegant to use a single protocol stack no matter where the destination of the message: Across a room or across an ocean, the TCP/IP overhead must be paid. This might have been a wise decision back then, especially given the unreliability of early Ethernet hardware, but it sets a high software overhead barrier for commercial systems of today. Such an obstacle lowers the enthusiasm for low-latency network interface hardware and low-latency interconnection networks if the software is just going to waste hundreds of microseconds when the message must travel only dozens of meters or less. It also can use significant processor resources. One rough rule of thumb is that each Mbit/sec of TCP/IP bandwidth needs about 1 MHz of processor speed, so a 1000-Mbit/sec link could saturate a processor with an 800- to 1000-MHz clock.

The flip side is that, from a software perspective, TCP/IP is the most desirable target since it is the most connected and, hence, provides the largest number of opportunities. The downside of using software optimized to a particular LAN or SAN is that it is limited. For example, communication from a Java program depends on TCP/IP, so optimization for another protocol would require creation of glue software to interface Java to it.

TCP/IP advocates point out that the protocol itself is theoretically not as burdensome as current implementations, but progress has been modest in commercial systems. There are also TCP/IP offloading engines in the market, with the hope of preserving the universal software model while reducing processor utilization and message latency. If processors continue to improve much faster than network speeds, or if multiple processors become ubiquitous, software TCP/IP may become less significant for processor utilization and message latency.

F.12 Concluding Remarks

Interconnection network design is one of the most exciting areas of computer architecture development today. With the advent of new multicore processor paradigms and advances in traditional multiprocessor/cluster systems and the Internet, many challenges and opportunities exist for interconnect architecture innovation. These apply to all levels of computer systems: communication between cores on a chip, between chips on a board, between boards in a system, and between computers in a machine room, over a local area and across the globe. Irrespective of their domain of application, interconnection networks should transfer the maximum amount of information within the least amount of time for given cost and power constraints so as not to bottleneck the system. Topology, routing, arbitration, switching, and flow control are among some of the key concepts in realizing such high-performance designs.

The design of interconnection networks is end-to-end: It includes injection links, reception links, and the interfaces at network end points as much as it does the topology, switches, and links within the network fabric. It is often the case that the bandwidth and overhead at the end node interfaces are the bottleneck, yet many mistakenly think of the interconnection network to mean only the network fabric. This is as bad as processor designers thinking of computer architecture to mean only the instruction set architecture or only the microarchitecture! End-to-end issues and understanding of the traffic characteristics make the design of interconnection networks challenging and very much relevant even today. For instance, the need for low end-to-end latency is driving the development of efficient network interfaces located closer to the processor/memory controller. We may soon see most multicore processors used in multiprocessor systems implementing network interfaces on-chip, devoting some core(s) to execute communication tasks. This is already the case for the IBM Blue Gene/L supercomputer, which uses one of its two cores on each processor chip for this purpose.

Networking has a long way to go from its humble shared-media beginnings. It is in "catch-up" mode, with switched-media point-to-point networks only recently displacing traditional bus-based networks in many networking domains, including on chip, I/O, and the local area. We are not near any performance plateaus, so we expect rapid advancement of WANs, LANs, SANs, and especially OCNs in the near future. Greater interconnection network performance is key to the information- and communication-centric vision of the future of our field, which, so far, has benefited many millions of people around the world in various ways. As the quotes at the beginning of this appendix suggest, this revolution in *two-way* communication is at the heart of changes in the form of our human associations and actions.

Acknowledgments

We express our sincere thanks to the following persons who, in some way, have contributed to the contents of the previous edition of the appendix: Lei Chai, Scott Clark, José Flich, Jose Manuel Garcia, Paco Gilabert, Rama Govindaraju, Manish Gupta, Wai Hong Ho, Siao Jer, Steven Keckler, Dhabaleswar (D.K.) Panda, Fabrizio Petrini, Steve Scott, Jeonghee Shin, Craig Stunkel, Sayantan Sur, Michael B. Taylor, and Bilal Zafar. We especially appreciate the new contributions of Jose Flich to this edition of the appendix.

F.13 Historical Perspective and References

This appendix has taken the perspective that interconnection networks for very different domains—from on-chip networks within a processor chip to wide area networks connecting computers across the globe—share many of the same concerns. With this, interconnection network concepts are presented in a unified way, irrespective of their application; however, their histories are vastly different, as evidenced by the different solutions adopted to address similar problems. The lack of significant interaction between research communities from the different domains certainly contributed to the diversity of implemented solutions. Highlighted below are relevant readings on each topic. In addition, good general texts featuring WAN and LAN networking have been written by Davie, Peterson, and Clark [1999] and by Kurose and Ross [2001]. Good texts focused on SANs for multiprocessors and clusters have been written by Duato, Yalamanchili, and Ni [2003] and by Dally and Towles [2004]. An informative chapter devoted to dead-lock resolution in interconnection networks was written by Pinkston [2004]. Finally, an edited work by Jantsch and Tenhunen [2003] on OCNs for multicore processors and system-on-chips is also interesting reading.

Wide Area Networks

Wide area networks are the earliest of the data interconnection networks. The forerunner of the Internet is the ARPANET, which in 1969 connected computer science departments across the United States that had research grants funded by the Advanced Research Project Agency (ARPA), a U.S. government agency. It was originally envisioned as using reliable communications at lower levels. Practical experience with failures of the underlying technology led to the failure-tolerant TCP/IP, which is the basis for the Internet today. Vint Cerf and Robert Kahn are credited with developing the TCP/IP protocols in the mid-1970s, winning the ACM Software Award in recognition of that achievement. Kahn [1972] is an early reference on the ideas of ARPANET. For those interested in learning more about TPC/IP, Stevens [1994–1996] has written classic books on the topic.

In 1975, there were roughly 100 networks in the ARPANET; in 1983, only 200. In 1995, the Internet encompassed 50,000 networks worldwide, about half of which were in the United States. That number is hard to calculate now, but the number of IP hosts grew by a factor of 15 from 1995 to 2000, reaching 100 million Internet hosts by the end of 2000. It has grown much faster since then. With most service providers assigning dynamic IP addresses, many local area networks using private IP addresses, and with most networks allowing wireless connections, the total number of hosts in the Internet is nearly impossible to compute. In July 2005, the Internet Systems Consortium (*www.isc.org*) estimated more than 350 million Internet hosts, with an annual increase of about 25% projected. Although key government networks made the Internet possible (i.e., ARPANET and NSFNET), these networks have been taken over by the commercial sector, allowing the Internet to thrive. But major innovations to the Internet are still likely to come from government-sponsored research projects rather than from the commercial sector. The National Science Foundation's Global Environment for Network Innovation (GENI) initiative is an example of this.

The most exciting application of the Internet is the World Wide Web, developed in 1989 by Tim Berners-Lee, a programmer at the European Center for Particle Research (CERN), for information access. In 1992, a young programmer at the University of Illinois, Marc Andreessen, developed a graphical interface for the Web called Mosaic. It became immensely popular. He later became a founder of Netscape, which popularized commercial browsers. In May 1995, at the time of the second edition of this book, there were over 30,000 Web pages, and the number was doubling every two months. During the writing of the third edition of this text, there were more than 1.3 billion Web pages. In December 2005, the number of Web servers approached 75 million, having increased by 30% during that same year.

Asynchronous Transfer Mode (ATM) was an attempt to design the definitive communication standard. It provided good support for data transmission as well as digital voice transmission (i.e., phone calls). From a technical point of view, it combined the best from packet switching and circuit switching, also providing excellent support for providing quality of service (QoS). Alles [1995] offers a good survey on ATM. In 1995, no one doubted that ATM was going to be the future for this community. Ten years later, the high equipment and personnel training costs basically killed ATM, and we returned back to the simplicity of TCP/IP. Another important blow to ATM was its defeat by the Ethernet family in the LAN domain, where packet switching achieved significantly lower latencies than ATM, which required establishing a connection before data transmission. ATM connectionless servers were later introduced in an attempt to fix this problem, but they were expensive and represented a central bottleneck in the LAN.

Finally, WANs today rely on optical fiber. Fiber technology has made so many advances that today WAN fiber bandwidth is often underutilized. The main reason for this is the commercial introduction of wavelength division multiplexing (WDM), which allows each fiber to transmit many data streams simultaneously over different wavelengths, thus allowing three orders of magnitude bandwidth increase in just one generation, that is, 3 to 5 years (a good text by Senior [1993] discusses optical fiber communications). However, IP routers may still become a bottleneck. At 10- to 40-Gbps link rates, and with thousands of ports in large core IP routers, packets must be processed very quickly—that is, within a few tens of nanoseconds. The most time-consuming operation is routing. The way IP addresses have been defined and assigned to Internet hosts makes routing very complicated, usually requiring a complex search in a tree structure for every packet. Network processors have become popular as a cost-effective solution for implementing routing and other packet-filtering operations. They usually are RISC-like and highly multithreaded and implement local stores instead of caches.

Local Area Networks

ARPA's success with wide area networks led directly to the most popular local area networks. Many researchers at Xerox Palo Alto Research Center had been funded by ARPA while working at universities, so they all knew the value of networking. In 1974, this group invented the Alto, the forerunner of today's desktop computers [Thacker et al. 1982], and the Ethernet [Metcalfe and Boggs 1976], today's LAN. This group—David Boggs, Butler Lampson, Ed McCreight, Bob Sprowl, and Chuck Thacker—became luminaries in computer science and engineering, collecting a treasure chest of awards among them.

This first Ethernet provided a 3-Mbit/sec interconnection, which seemed like an unlimited amount of communication bandwidth with computers of that era. It relied on the interconnect technology developed for the cable television industry. Special microcode support gave a round-trip time of 50 μs for the Alto over Ethernet, which is still a respectable latency. It was Boggs' experience as a ham radio operator that led to a design that did not need a central arbiter, but instead listened before use and then varied back-off times in case of conflicts.

The announcement by Digital Equipment Corporation, Intel, and Xerox of a standard for 10-Mbit/sec Ethernet was critical to the commercial success of

Ethernet. This announcement short-circuited a lengthy IEEE standards effort, which eventually did publish IEEE 802.3 as a standard for Ethernet.

There have been several unsuccessful candidates that have tried to replace the Ethernet. The Fiber Data Distribution Interconnect (FDDI) committee, unfortunately, took a very long time to agree on the standard, and the resulting interfaces were expensive. It was also a shared medium when switches were becoming affordable. ATM also missed the opportunity in part because of the long time to standardize the LAN version of ATM, and in part because of the high latency and poor behavior of ATM connectionless servers, as mentioned above. InfiniBand for the reasons discussed below has also faltered. As a result, Ethernet continues to be the absolute leader in the LAN environment, and it remains a strong opponent in the high-performance computing market as well, competing against the SANs by delivering high bandwidth at low cost. The main drawback of Ethernet for high-end systems is its relatively high latency and lack of support in most interface cards to implement the necessary protocols.

Because of failures of the past, LAN modernization efforts have been centered on extending Ethernet to lower-cost media such as unshielded twisted pair (UTP), switched interconnects, and higher link speeds as well as to new domains such as wireless communication. Practically all new PC motherboards and laptops implement a Fast/Gigabit Ethernet port (100/1000 Mbps), and most laptops implement a 54 Mbps Wireless Ethernet connection. Also, home wired or wireless LANs connecting all the home appliances, set-top boxes, desktops, and laptops to a shared Internet connection are very common. Spurgeon [2006] has provided a nice online summary of Ethernet technology, including some of its history.

System Area Networks

One of the first nonblocking multistage interconnection networks was proposed by Clos [1953] for use in telephone exchange offices. Building on this, many early inventions for system area networks came from their use in massively parallel processors (MPPs). One of the first MPPs was the Illiac IV, a SIMD array built in the early 1970s with 64 processing elements (“massive” at that time) interconnected using a topology based on a 2D torus that provided neighbor-to-neighbor communication. Another representative of early MPP was the Cosmic Cube, which used Ethernet interface chips to connect 64 processors in a 6-cube. Communication between nonneighboring nodes was made possible by store-and-forwarding of packets at intermediate nodes toward their final destination. A much larger and truly “massive” MPP built in the mid-1980s was the Connection Machine, a SIMD multiprocessor consisting of 64 K 1-bit processing elements, which also used a hypercube with store-and-forwarding. Since these early MPP machines, interconnection networks have improved considerably.

In the 1970s through the 1990s, considerable research went into trying to optimize the topology and, later, the routing algorithm, switching, arbitration, and flow control techniques. Initially, research focused on maximizing performance with little attention paid to implementation constraints or crosscutting issues. Many exotic topologies were proposed having very interesting properties, but most of them complicated the routing. Rising from the fray was the hypercube, a very popular network in the 1980s that has all but disappeared from MPPs since the 1990s. What contributed to this shift was a performance model by Dally [1990] that showed that if the implementation is wire limited, lower-dimensional topologies achieve better performance than higher-dimensional ones because of their wider links for a given wire budget. Many designers followed that trend assuming their designs to be wire limited, even though most implementations were (and still are) pin limited. Several supercomputers since the 1990s have implemented low-dimensional topologies, including the Intel Paragon, Cray T3D, Cray T3E, HP AlphaServer, Intel ASCI Red, and IBM Blue Gene/L.

Meanwhile, other designers followed a very different approach, implementing bidirectional MINs in order to reduce the number of required switches below the number of network nodes. The most popular bidirectional MIN was the fat tree topology, originally proposed by Leiserson [1985] and first used in the Connection Machine CM-5 supercomputer and, later, the IBM ASCI White and ASC Purple supercomputers. This indirect topology was also used in several European parallel computers based on the Transputer. The Quadrics network has inherited characteristics from some of those Transputer-based networks. Myrinet has also evolved significantly from its first version, with Myrinet 2000 incorporating the fat tree as its principal topology. Indeed, most current implementations of SANs, including Myrinet, InfiniBand, and Quadrics as well as future implementations such as PCI-Express Advanced Switching, are based on fat trees.

Although the topology is the most visible aspect of a network, other features also have a significant impact on performance. A seminal work that raised awareness of deadlock properties in computer systems was published by Holt [1972]. Early techniques for avoiding deadlock in store-and-forward networks were proposed by Merlin and Schweitzer [1980] and by Gunther [1981]. Pipelined switching techniques were first introduced by Kermani and Kleinrock [1979] (virtual cut-through) and improved upon by Dally and Seitz [1986] (wormhole), which significantly reduced low-load latency and the topology's impact on message latency over previously proposed techniques. Wormhole switching was initially better than virtual cut-through largely because flow control could be implemented at a granularity smaller than a packet, allowing high-bandwidth links that were not as constrained by available switch memory bandwidth. Today, virtual cut-through is usually preferred over wormhole because it achieves higher throughput due to less HOL blocking effects and is enabled by current integration technology that allows the implementation of many packet buffers per link.

Tamir and Frazier [1992] laid the groundwork for virtual output queuing with the notion of dynamically allocated multiqueues. Around this same time, Dally [1992] contributed the concept of virtual channels, which was key to the development of more efficient deadlock-free routing algorithms and congestion-reducing flow control techniques for improved network throughput. Another highly relevant contribution to routing was a new theory proposed by Duato [1993] that allowed the implementation of fully adaptive routing with just one "escape" virtual channel to avoid deadlock. Previous to this, the required number of virtual channels to avoid deadlock increased exponentially with the number of network dimensions. Pinkston and Warnakulasuriya [1997] went on to show that deadlock actually can occur very infrequently, giving credence to deadlock recovery routing approaches. Scott and Goodman [1994] were among the first to analyze the usefulness of pipelined channels for making link bandwidth independent of the time of flight. These and many other innovations have become quite popular, finding use in most high-performance interconnection networks, both past and present. The IBM Blue Gene/L, for example, implements virtual cut-through switching, four virtual channels per link, fully adaptive routing with one escape channel, and pipelined links.

MPPs represent a very small (and currently shrinking) fraction of the information technology market, giving way to bladed servers and clusters. In the United States, government programs such as the Advanced Simulation and Computing (ASC) program (formerly known as the Accelerated Strategic Computing Initiative, or ASCI) have promoted the design of those machines, resulting in a series of increasingly powerful one-of-a-kind MPPs costing $50 million to $100 million. These days, many are basically lower-cost clusters of symmetric multiprocessors (SMPs) (see Pfister [1998] and Sterling [2001] for two perspectives on clustering). In fact, in 2005, nearly 75% of the TOP500 supercomputers were clusters. Nevertheless, the design of each generation of MPPs and even clusters pushes interconnection network research forward to confront new problems arising due to shear size and other scaling factors. For instance, source-based routing—the simplest form of routing—does not scale well to large systems. Likewise, fat trees require increasingly longer links as the network size increases, which led IBM Blue Gene/L designers to adopt a 3D torus network with distributed routing that can be implemented with bounded-length links.

Storage Area Networks

System area networks were originally designed for a single room or single floor (thus their distances are tens to hundreds of meters) and were for use in MPPs and clusters. In the intervening years, the acronym SAN has been co-opted to also mean storage area networks, whereby networking technology is used to connect storage devices to compute servers. Today, many refer to "storage" when they say SAN. The most widely used SAN example in 2006 was Fibre Channel (FC), which comes in many varieties, including various versions of Fibre Channel Arbitrated Loop (FC-AL) and Fibre Channel Switched (FC-SW). Not only are disk arrays attached to servers via FC links, but there are even some disks with FC links attached to switches so that storage area networks can enjoy the benefits of greater bandwidth and interconnectivity of switching.

In October 2000, the InfiniBand Trade Association announced the version 1.0 specification of InfiniBand [InfiniBand Trade Association 2001]. Led by Intel, HP, IBM, Sun, and other companies, it was targeted to the high-performance computing market as a successor to the PCI bus by having point-to-point links and switches with its own set of protocols. Its characteristics are desirable potentially both for system area networks to connect clusters and for storage area networks to connect disk arrays to servers. Consequently, it has had strong competition from both fronts. On the storage area networking side, the chief competition for InfiniBand has been the rapidly improving Ethernet technology widely used in LANs. The Internet Engineering Task Force proposed a standard called iSCSI to send SCSI commands over IP networks [Satran et al. 2001]. Given the cost advantages of the higher-volume Ethernet switches and interface cards, Gigabit Ethernet dominates the low-end and medium range for this market. What's more, the slow introduction of InfiniBand and its small market share delayed the development of chip sets incorporating native support for InfiniBand. Therefore, network interface cards had to be plugged into the PCI or PCI-X bus, thus never delivering on the promise of replacing the PCI bus.

It was another I/O standard, PCI-Express, that finally replaced the PCI bus. Like InfiniBand, PCI-Express implements a switched network but with point-to-point serial links. To its credit, it maintains software compatibility with the PCI bus, drastically simplifying migration to the new I/O interface. Moreover, PCI-Express benefited significantly from mass market production and has found application in the desktop market for connecting one or more high-end graphics cards, making gamers very happy. Every PC motherboard now implements one or more 16x PCI-Express interfaces. PCI-Express absolutely dominates the I/O interface, but the current standard does not provide support for interprocessor communication.

Yet another standard, Advanced Switching Interconnect (ASI), may emerge as a complementary technology to PCI-Express. ASI is compatible with PCI-Express, thus linking directly to current motherboards, but it also implements support for interprocessor communication as well as I/O. Its defenders believe that it will eventually replace both SANs and LANs with a unified network in the data center market, but ironically this was also said of InfiniBand. The interested reader is referred to Pinkston et al. [2003] for a detailed discussion on this. There is also a new disk interface standard called Serial Advanced Technology Attachment (SATA) that is replacing parallel Integrated Device Electronics (IDE) with serial signaling technology to allow for increased bandwidth. Most disks in the market use this new interface, but keep in mind that Fibre Channel is still alive and well. Indeed, most of the promises made by InfiniBand in the SAN market were satisfied by Fibre Channel first, thus increasing their share of the market.

Some believe that Ethernet, PCI-Express, and SATA have the edge in the LAN, I/O interface, and disk interface areas, respectively. But the fate of the remaining storage area networking contenders depends on many factors. A wonderful characteristic of computer architecture is that such issues will not remain endless academic debates, unresolved as people rehash the same arguments repeatedly. Instead, the battle is fought in the marketplace, with well-funded and talented groups giving their best efforts at shaping the future. Moreover, constant changes to technology reward those who are either astute or lucky. The best combination of technology and follow-through has often determined commercial success. Time will tell us who will win and who will lose, at least for the next round!

On-Chip Networks

Relative to the other network domains, on-chip networks are in their infancy. As recently as the late 1990s, the traditional way of interconnecting devices such as caches, register files, ALUs, and other functional units within a chip was to use dedicated links aimed at minimizing latency or shared buses aimed at simplicity. But with subsequent increases in the volume of interconnected devices on a single chip, the length and delay of wires to cross a chip, and chip power consumption, it has become important to share on-chip interconnect bandwidth in a more structured way, giving rise to the notion of a network on-chip. Among the first to recognize this were Agarwal [Waingold et al. 1997] and Dally [Dally 1999; Dally and Towles 2001]. They and others argued that on-chip networks that route packets allow efficient sharing of burgeoning wire resources between many communication flows and also facilitate modularity to mitigate chip-crossing wire delay problems identified by Ho, Mai, and Horowitz [2001]. Switched on-chip networks were also viewed as providing better fault isolation and tolerance. Challenges in designing these networks were later described by Taylor et al. [2005], who also proposed a 5-tuple model for characterizing the delay of OCNs. A design process for OCNs that provides a complete synthesis flow was proposed by Bertozzi et al. [2005]. Following these early works, much research and development has gone into on-chip network design, making this a very hot area of microarchitecture activity.

Multicore and tiled designs featuring on-chip networks have become very popular since the turn of the millennium. Pinkston and Shin [2005] provide a survey of on-chip networks used in early multicore/tiled systems. Most designs exploit the reduced wiring complexity of switched OCNs as the paths between cores/tiles can be precisely defined and optimized early in the design process, thus enabling improved power and performance characteristics. With typically tens of thousands of wires attached to the four edges of a core or tile as "pinouts," wire resources can be traded off for improved network performance by having very wide channels over which data can be sent broadside (and possibly scaled up or down according to the power management technique), as opposed to serializing the data over fixed narrow channels.

Rings, meshes, and crossbars are straightforward to implement in planar chip technology and routing is easily defined on them, so these were popular topological choices in early switched OCNs. It will be interesting to see if this trend continues in the future when several tens to hundreds of heterogeneous cores and tiles will likely be interconnected within a single chip, possibly using 3D integration technology. Considering that processor microarchitecture has evolved significantly from its early beginnings in response to application demands and technological advancements, we would expect to see vast architectural improvements to on-chip networks as well.

References

Agarwal, A., 1991. Limits on interconnection network performance. IEEE Trans. on Parallel and Distributed Systems 2 (4 (April)), 398–412.

Alles, A., 1995. “ATM internetworking” (May). www.cisco.com/warp/public/614/12.html.

Anderson, T.E., Culler, D.E., Patterson, D., 1995. A case for NOW (networks of workstations). IEEE Micro 15 (1 (February)), 54–64.

Anjan, K.V., Pinkston, T.M., 1995. An efficient, fully-adaptive deadlock recovery scheme: Disha. In: Proc. 22nd Annual Int’l. Symposium on Computer Architecture, June 22–24, 1995. Santa Margherita Ligure, Italy.

Arpaci, R.H., Culler, D.E., Krishnamurthy, A., Steinberg, S.G., Yelick, K., 1995. Empirical evaluation of the Cray-T3D: A compiler perspective. In: Proc. 22nd Annual Int’l. Symposium on Computer Architecture, June 22–24, 1995. Santa Margherita Ligure, Italy.

Bell, G., Gray, J., 2001. Crays, Clusters and Centers. Microsoft Corporation, Redmond, Wash. MSR-TR-2001-76.

Benes, V.E., 1962. Rearrangeable three stage connecting networks. Bell Syst. Tech. J. 41, 1481–1492.

Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., De Micheli, G., 2005. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Trans. on Parallel and Distributed Systems 16 (2 (February)), 113–130.

Bhuyan, L.N., Agrawal, D.P., 1984. Generalized hypercube and hyperbus structures for a computer network. IEEE Trans. on Computers 32 (4 (April)), 322–333.

Brewer, E.A., Kuszmaul, B.C., 1994. How to get good performance from the CM-5 data network. In: Proc. Eighth Int’l Parallel Processing Symposium, April 26–29, 1994. Cancun, Mexico.

Clos, C., 1953. A study of non-blocking switching networks. Bell Systems Technical Journal 32 (March), 406–424.

Dally, W.J., 1990. Performance analysis of k-ary n-cube interconnection networks. IEEE Trans. on Computers 39 (6 (June)), 775–785.

Dally, W.J., 1992. Virtual channel flow control. IEEE Trans. on Parallel and Distributed Systems 3 (2 (March)), 194–205.

Dally, W.J., 1999. Interconnect limited VLSI architecture. In: Proc. of the Int’l. Interconnect Technology Conference, May 24–26, 1999. San Francisco, Calif.

Dally, W.J., Seitz, C.I., 1986. The torus routing chip. Distributed Computing 1 (4), 187–196.

Dally, W.J., Towles, B., 2001. Route packets, not wires: On-chip interconnection networks. In: Proc. of the 38th Design Automation Conference, June 18–22, 2001. Las Vegas, Nev.

Dally, W.J., Towles, B., 2004. Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers, San Francisco.

Davie, B.S., Peterson, L.L., Clark, D., 1999. Computer Networks: A Systems Approach, second ed. Morgan Kaufmann Publishers, San Francisco.

Duato, J., 1993. A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Trans. on Parallel and Distributed Systems 4 (12 (December)), 1320–1331.

Duato, J., Pinkston, T.M., 2001. A general theory for deadlock-free adaptive routing using a mixed set of resources. IEEE Trans. on Parallel and Distributed Systems 12 (12 (December)), 1219–1235.

Duato, J., Yalamanchili, S., Ni, L., 2003. Interconnection Networks: An Engineering Approach. Morgan Kaufmann Publishers, San Francisco. 2nd printing.

Duato, J., Johnson, I., Flich, J., Naven, F., Garcia, P., Nachiondo, T., 2005a. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In: Proc. 11th Int’l. Symposium on High Performance Computer Architecture, February 12–16, 2005 San Francisco.

Duato, J., Lysne, O., Pang, R., Pinkston, T.M., 2005b. Part I: A theory for deadlock-free dynamic reconfiguration of interconnection networks. IEEE Trans. on Parallel and Distributed Systems 16 (5 (May)), 412–427.

Flich, J., Bertozzi, D., 2010. Designing Network-on-Chip Architectures in the Nanoscale Era. CRC Press, Boca Raton, FL.

Glass, C.J., Ni, L.M., 1992. The Turn Model for adaptive routing. In: Proc. 19th Int’l. Symposium on Computer Architecture. May, Gold Coast, Australia.

Gunther, K.D., 1981. Prevention of deadlocks in packet-switched data transport systems. IEEE Trans. on Communications, 512–524. COM–29:4 (April).

Ho, R., Mai, K.W., Horowitz, M.A., 2001. The future of wires. In: Proc. of the IEEE 89:4 (April), pp. 490–504.

Holt, R.C., 1972. Some deadlock properties of computer systems. ACM Computer Surveys 4 (3 (September)), 179–196.

Hoskote, Y., Vangal, S., Singh, A., Borkar, N., Borkar, S., 2007. A 5-ghz mesh interconnect for a teraflops processor. IEEE Micro 27 (5), 51–61.

Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, S., Ruhl, G., Jenkins, D., Wilson, H., Borka, N., Schrom, G., Pailet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R., Mattson, T., 2010. A 48-core IA-32 message-passing processor with DVFS in 45 nm CMOS. In: IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp. 58–59.

InfiniBand Trade Association, 2001. InfiniBand Architecture Specifications Release 1.0.a. www.infinibandta.org.

Jantsch, A., Tenhunen, H. (Eds.), 2003. Networks on Chips. Kluwer Academic Publishers, The Netherlands.

Kahn, R.E., 1972. Resource-sharing computer communication networks. In: Proc. IEEE 60:11 (November), pp. 1397–1407.

Kermani, P., Kleinrock, L., 1979. Virtual cut-through: A new computer communication switching technique. Computer Networks 3 (January), 267–286.

Kurose, J.F., Ross, K.W., 2001. Computer Networking: A Top-Down Approach Featuring the Internet. Addison-Wesley, Boston.

Leiserson, C.E., 1985. Fat trees: Universal networks for hardware-efficient supercomputing. IEEE Trans. on Computers, 892–901. C–34:10 (October).

Merlin, P.M., Schweitzer, P.J., 1980. Deadlock avoidance in store-and-forward networks. I. Store-and-forward deadlock. IEEE Trans. on Communications, 345–354. COM–28:3 (March).

Metcalfe, R.M., 1993. Computer/network interface design: Lessons from Arpanet and Ethernet. IEEE J. on Selected Areas in Communications 11 (2 (February)), 173–180.

Metcalfe, R.M., Boggs, D.R., 1976. Ethernet: Distributed packet switching for local computer networks. Comm. ACM 19 (7 (July)), 395–404.

Partridge, C., 1994. Gigabit Networking. Addison-Wesley, Reading, Mass.

Peh, L.S., Dally, W.J., 2001. A delay model and speculative architecture for pipelined routers. In: Proc. 7th Int'l. Symposium on High Performance Computer Architecture, January 20–24, 2001. Monterrey, Mexico.

Pfister, G.F., 1998. In Search of Clusters, second ed. Prentice Hall, Upper Saddle River, N.J.

Pinkston, T.M., 2004. Deadlock characterization and resolution in interconnection networks. In: Zhu, M.C., Fanti, M.P. (Eds.), Deadlock Resolution in Computer-Integrated Systems. CRC Press, Boca Raton, Fl, pp. 445–492.

Pinkston, T.M., Shin, J., 2005. Trends toward on-chip networked microsystems. Int'l. J. of High Performance Computing and Networking 3 (1), 3–18.

Pinkston, T.M., Warnakulasuriya, S., 1997. On deadlocks in interconnection networks. In: Proc. 24th Int'l. Symposium on Computer Architecture, June 2–4, 1997. Denver, Colo.

Pinkston, T.M., Benner, A., Krause, M., Robinson, I., Sterling, T., 2003. InfiniBand: The 'de facto' future standard for system and local area networks or just a scalable replacement for PCI buses?" Special Issue on Communication Architecture for Clusters 6:2 (April). Cluster Computing, 95–104.

Puente, V., Beivide, R., Gregorio, J.A., Prellezo, J.M., Duato, J., Izu, C., 1999. Adaptive bubble router: A design to improve performance in torus networks. In: Proc. 28th Int'l. Conference on Parallel Processing, September 21–24, 1999. Aizu-Wakamatsu, Japan.

Rodrigo, S., Flich, J., Duato, J., Hummel, M., 2008. Efficient unicast and multicast support for CMPs. In: Proc. 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41), November 8–12, 2008. Lake Como, Italy, pp. 364–375.

Saltzer, J.H., Reed, D.P., Clark, D.D., 1984. End-to-end arguments in system design. ACM Trans. on Computer Systems 2 (4 (November)), 277–288.

Satran, J., Smith, D., Meth, K., Sapuntzakis, C., Wakeley, M., Von Stamwitz, P., Haagens, R., Zeidner, E., Dalle Ore, L., Klein, Y., 2001. "iSCSI", IPS working group of IETF, Internet draft. www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt.

Scott, S.L., Goodman, J., 1994. The impact of pipelined channels on k-ary n-cube networks. IEEE Trans. on Parallel and Distributed Systems 5 (1 (January)), 1–16.

Senior, J.M., 1993. Optical Fiber Commmunications: Principles and Practice, second ed. Prentice Hall, Hertfordshire, U.K..

Spurgeon, C., 2006. Charles Spurgeon's Ethernet Web Site. www.etherman-age.com/ethernet/ethernet.html.

Sterling, T., 2001. Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux. MIT Press, Cambridge, Mass.
Stevens, W.R., 1994–1996. *TCP/IP Illustrated* (three volumes). Addison-Wesley, Reading, Mass.
Tamir, Y., Frazier, G., 1992. Dynamically-allocated multi-queue buffers for VLSI communication switches. IEEE Trans. on Computers 41 (6 (June)), 725–734.
Tanenbaum, A.S., 1988. Computer Networks, second ed. Prentice Hall, Englewood Cliffs, N.J.
Taylor, M.B., Lee, W., Amarasinghe, S.P., Agarwal, A., 2005. Scalar operand networks. IEEE Trans. on Parallel and Distributed Systems 16 (2 (February)), 145–162.
Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., Boggs, D.R., 1982. Alto: A personal computer. In: Siewiorek, D.P., Bell, C.G., Newell, A. (Eds.), Computer Structures: Principles and Examples. McGraw-Hill, New York, pp. 549–572.
TILE-GX, http://www.tilera.com/sites/default/files/productbriefs/PB025_TILE-Gx_Processor_A_v3.pdf.
Vaidya, A.S., Sivasubramaniam, A., Das, C.R., 1997. Performance benefits of virtual channels and adaptive routing: An application-driven study. In: Proc. 11th ACM Int'l Conference on Supercomputing, July 7–11, 1997. Vienna, Austria.
Van Leeuwen, J., Tan, R.B., 1987. Interval Routing. The Computer Journal 30 (4), 298–307.
von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E., 1992. Active messages: A mechanism for integrated communication and computation. In: Proc. 19th Annual Int'l. Symposium on Computer Architecture, May 19–21, 1992. Gold Coast, Australia.
Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A., 1997. Baring it all to software: Raw Machines. IEEE Computer 30 (September), 86–93.
Yang, Y., Mason, G., 1991. Nonblocking broadcast switching networks. IEEE Trans. on Computers 40 (9 (September)), 1005–1015.

Exercises

Solutions to "starred" exercises are available for instructors who register at *textbooks.elsevier.com.*

✪ F.1 [15] < F.2, F.3 > Is electronic communication always faster than nonelectronic means for longer distances? Calculate the time to send 1000 GB using 25 8-mm tapes and an overnight delivery service versus sending 1000 GB by FTP over the Internet. Make the following four assumptions:

- The tapes are picked up at 4 P.M. Pacific time and delivered 4200 km away at 10 A.M. Eastern time (7 A.M. Pacific time).
- On one route the slowest link is a T3 line, which transfers at 45 Mbits/sec.
- On another route the slowest link is a 100-Mbit/sec Ethernet.
- You can use 50% of the slowest link between the two sites.

Will all the bytes sent by either Internet route arrive before the overnight delivery person arrives?

✪ F.2 [10] < F.2, F.3 > For the same assumptions as Exercise F.1, what is the bandwidth of overnight delivery for a 1000-GB package?

✪ F.3 [10] < F.2, F.3 > For the same assumptions as Exercise F.1, what is the minimum bandwidth of the slowest link to beat overnight delivery? What standard network options match that speed?

✪ F.4 [15] < F.2, F.3 > The original Ethernet standard was for 10 Mbits/sec and a maximum distance of 2.5 km. How many bytes could be in flight in the original Ethernet? Assume you can use 90% of the peak bandwidth.

✪ F.5 [15] < F.2, F.3 > Flow control is a problem for WANs due to the long time of flight, as the example on page F-14 illustrates. Ethernet did not include flow control when it was first standardized at 10 Mbits/sec. Calculate the number of bytes in flight for a 10-Gbit/sec Ethernet over a 100 meter link, assuming you can use 90% of peak bandwidth. What does your answer mean for network designers?

✪ F.6 [15] < F.2, F.3 > Assume the total overhead to send a zero-length data packet on an Ethernet is 100 μs and that an unloaded network can transmit at 90% of the peak 1000-Mbit/sec rating. For the purposes of this question, assume that the size of the Ethernet header and trailer is 56 bytes. Assume a continuous stream of packets of the same size. Plot the delivered bandwidth of user data in Mbits/sec as the payload data size varies from 32 bytes to the maximum size of 1500 bytes in 32-byte increments.

✪ F.7 [10] < F.2, F.3 > Exercise F.6 suggests that the delivered Ethernet bandwidth to a single user may be disappointing. Making the same assumptions as in that exercise, by how much would the maximum payload size have to be increased to deliver half of the peak bandwidth?

✪ F.8 [10] < F.2, F.3 > One reason that ATM has a fixed transfer size is that when a short message is behind a long message, a node may need to wait for an entire transfer to complete. For applications that are time sensitive, such as when transmitting voice or video, the large transfer size may result in transmission delays that are too long for the application. On an unloaded interconnection, what is the worstcase delay in microseconds if a node must wait for one full-size Ethernet packet versus an ATM transfer? See Figure F.30 (page F-78) to find the packet sizes. For this question assume that you can transmit at 100% of the 622-Mbits/sec ATM network and 100% of the 1000-Mbit/sec Ethernet.

✪ F.9 [10] < F.2, F.3 > Exercise F.7 suggests the need for expanding the maximum pay-load to increase the delivered bandwidth, but Exercise F.8 suggests the impact on worst-case latency of making it longer. What would be the impact on latency of increasing the maximum payload size by the answer to Exercise F.7?

✪ F.10 [12/12/20] < F.4 > The Omega network shown in Figure F.11 on page F-31 consists of three columns of four switches, each with two inputs and two outputs. Each switch can be set to *straight*, which connects the upper switch input to the upper switch output and the lower input to the lower output, and to *exchange,* which connects the upper input to the lower output and *vice versa* for the lower input. For each column of switches, label the inputs and outputs 0, 1, …, 7 from top to bottom, to correspond with the numbering of the processors.

a. [12] < F.4 > When a switch is set to exchange and a message passes through, what is the relationship between the label values for the switch input and output used by the message? (*Hint*: Think in terms of operations on the digits of the binary representation of the label number.)

b. [12] < F.4 > Between any two switches in adjacent columns that are connected by a link, what is the relationship between the label of the output connected to the input?

c. [20] < F.4 > Based on your results in parts (a) and (b), design and describe a simple routing scheme for distributed control of the Omega network. A message will carry a *routing tag* computed by the sending processor. Describe how the processor computes the tag and how each switch can set itself by examining a bit of the routing tag.

✪ **F.11** [12/12/12/12/12/12] < F.4 > Prove whether or not it is possible to realize the following permutations (i.e., communication patterns) on the eight-node Omega network shown in Figure F.11 on page F-31:

a. [12] < F.4 > Bit-reversal permutation—the node with binary coordinates a_{n-1}, a_{n-2}, …, a_1, a_0 communicates with the node a_0, a_1, …, a_{n-2}, a_{n-1}.

b. [12] < F.4 > Perfect shuffle permutation—the node with binary coordinates a_{n-1}, a_{n-2}, …, a_1, a_0 communicates with the node a_{n-2}, a_{n-3}, …, a_0, a_{n-1} (i.e., rotate left 1 bit).

c. [12] < F.4 > Bit-complement permutation—the node with binary coordinates a_{n-1}, a_{n-2}, …, a_1, a_0 communicates with the node $\overline{a}_{n-1}$, $\overline{a}_{n-2}$,…, $\overline{a}_1$, $\overline{a}_0$ (i.e., complement each bit).

d. [12] < F.4 > Butterfly permutation—the node with binary coordinates a_{n-1}, a_{n-2}, …, a_1, a_0 communicates with the node a_0, a_{n-2}, …, a_1, a_{n-1} (i.e., swap the most and least significant bits).

e. [12] < F.4 > Matrix transpose permutation—the node with binary coordinates a_{n-1}, a_{n-2}, …, a_1, a_0 communicates with the node $a_{n/2-1}$, …, a_0, a_{n-1}, …, $a_{n/2}$ (i.e., transpose the bits in positions approximately halfway around).

f. [12] < F.4 > Barrel-shift permutation—node i communicates with node $i+1$ modulo $N-1$, where N is the total number of nodes and $0 \leq i$.

✪ **F.12** [12] < F.4 > Design a network topology using 18-port crossbar switches that has the minimum number of switches to connect 64 nodes. Each switch port supports communication to and from one device.

✪ **F.13** [15] < F.4 > Design a network topology that has the minimum latency through the switches for 64 nodes using 18-port crossbar switches. Assume unit delay in the switches and zero delay for wires.

✪ **F.14** [15] < F.4 > Design a switch topology that balances the bandwidth required for all links for 64 nodes using 18-port crossbar switches. Assume a uniform traffic pattern.

✪ F.15 [15] < F.4 > Compare the interconnection latency of a crossbar, Omega network, and fat tree with eight nodes. Use Figure F.11 on page F-31, Figure F.12 on page F-33, and Figure F.14 on page F-37. Assume that the fat tree is built entirely from two-input, two-output switches so that its hardware resources are more comparable to that of the Omega network. Assume that each switch costs a unit time delay. Assume that the fat tree randomly picks a path, so give the best case and worst case for each example. How long will it take to send a message from node 0 to node 6? How long will it take node 1 and node 7 to communicate?

✪ F.16 [15] < F.4 > Draw the topology of a 6-cube after the same manner of the 4-cube in Figure F.14 on page F-37. What is the maximum and average number of hops needed by packets assuming a uniform distribution of packet destinations?

✪ F.17 [15] < F.4 > Complete a table similar to Figure F.15 on page F-40 that captures the performance and cost of various network topologies, but do it for the general case of N nodes using $k \times k$ switches instead of the specific case of 64 nodes.

✪ F.18 [20] < F.4 > Repeat the example given on page F-41, but use the bit-complement communication pattern given in Exercise F.11 instead of NEWS communication.

✪ F.19 [15] < F.5 > Give the four specific conditions necessary for deadlock to exist in an interconnection network. Which of these are removed by dimension-order routing? Which of these are removed in adaptive routing with the use of "escape" routing paths? Which of these are removed in adaptive routing with the technique of deadlock recovery (regressive or progressive)? Explain your answer.

✪ F.20 [12/12/12/12] < F.5 > Prove whether or not the following routing algorithms based on prohibiting dimensional turns are suitable to be used as escape paths for 2D meshes by analyzing whether they are both connected and deadlock-free. Explain your answer. (*Hint*: You may wish to refer to the Turn Model algorithm and/or to prove your answer by drawing a directed graph for a 4 × 4 mesh that depicts dependencies between channels and verifying the channel dependency graph is free of cycles.) The routing algorithms are expressed with the following abbreviations: W = west, E = east, N = north, and S = south.

a. [12] < F.5 > Allowed turns are from W to N, E to N, S to W, and S to E.

b. [12] < F.5 > Allowed turns are from W to S, E to S, N to E, and S to E.

c. [12] < F.5 > Allowed turns are from W to S, E to S, N to W, S to E, W to N, and S to W.

d. [12] < F.5 > Allowed turns are from S to E, E to S, S to W, N to W, N to E, and E to N.

✪ F.21 [15] < F.5 > Compute and compare the upper bound for the efficiency factor, ρ, for dimension-order routing and up*/down* routing assuming uniformly distributed traffic on a 64-node 2D mesh network. For up*/down* routing, assume optimal placement of the root node (i.e., a node near the middle of the mesh). (*Hint*: You will have to find the loading of links across the network bisection that carries the global load as determined by the routing algorithm.)

✪ F.22 [15] <F.5> For the same assumptions as Exercise F.21, find the efficiency factor for up*/down* routing on a 64-node fat tree network using 4×4 switches. Compare this result with the ρ found for up*/down* routing on a 2D mesh. Explain.

✪ F.23 [15] <F.5> Calculate the probability of matching two-phased arbitration requests from all *k* input ports of a switch simultaneously to the *k* output ports assuming a uniform distribution of requests and grants to/from output ports. How does this compare to the matching probability for three-phased arbitration in which each of the *k* input ports can make two simultaneous requests (again, assuming a uniform random distribution of requests and grants)?

✪ F.24 [15] <F.5> The equation on page F-52 shows the value of cut-through switching. Ethernet switches used to build clusters often do not support cut-through switching. Compare the time to transfer 1500 bytes over a 1000-Mbit/sec Ethernet with and without cut-through switching for a 64-node cluster. Assume that each Ethernet switch takes 1.0 μs and that a message goes through seven intermediate switches.

✪ F.25 [15] <F.5> Making the same assumptions as in Exercise F.24, what is the difference between cut-through and store-and-forward switching for 32 bytes?

✪ F.26 [15] <F.5> One way to reduce latency is to use larger switches. Unlike Exercise F.24, let's assume we need only three intermediate switches to connect any two nodes in the cluster. Make the same assumptions as in Exercise F.24 for the remaining parameters. What is the difference between cut-through and store-and-forward for 1500 bytes? For 32 bytes?

✪ F.27 [20] <F.5> Using FlexSim 1.2 (*http://ceng.usc.edu/smart/FlexSim/flexsim.html*) or some other cycle-accurate network simulator, simulate a 256-node 2D torus network assuming wormhole routing, 32-flit packets, uniform (random) communication pattern, and four virtual channels. Compare the performance of deterministic routing using DOR, adaptive routing using escape paths (i.e., Duato's Protocol), and true fully adaptive routing using progressive deadlock recovery (i.e., Disha routing). Do so by plotting latency versus applied load and through-put versus applied load for each, as is done in Figure F.19 for the example on page F-53. Also run simulations and plot results for two and eight virtual channels for each. Compare and explain your results by addressing how/why the number and use of virtual channels by the various routing algorithms affect network performance. (*Hint*: Be sure to let the simulation reach steady state by allowing a warm-up period of a several thousand network cycles before gathering results.)

✪ F.28 [20] <F.5> Repeat Exercise F.27 using bit-reversal communication instead of the uniform random communication pattern. Compare and explain your results by addressing how/why the communication pattern affects network performance.

✪ F.29 [40] <F.5> Repeat Exercises F.27 and F.28 using 16-flit packets and 128-flit packets. Compare and explain your results by addressing how/why the packet size along with the other design parameters affect network performance.

F.30 [20] <F.2, F.4, F.5, F.8> Figures F.7, F.16, and F.20 show interconnection network characteristics of several of the top 500 supercomputers by machine type as of the publication of the fourth edition. Update that figure to the most recent top 500. How have the systems and their networks changed since the data in the original figure? Do similar comparisons for OCNs used in microprocessors and SANs targeted for clusters using Figures F.29 and F.31.

✪ F.31 [12/12/12/15/15/18] <F.8> Use the M/M/1 queuing model to answer this exercise. Measurements of a network bridge show that packets arrive at 200 packets per second and that the gateway forwards them in about 2 ms.

a. [12] <F.8> What is the utilization of the gateway?

b. [12] <F.8> What is the mean number of packets in the gateway?

c. [12] <F.8> What is the mean time spent in the gateway?

d. [15] <F.8> Plot response time versus utilization as you vary the arrival rate.

e. [15] <F.8> For an M/M/1 queue, the probability of finding n or more tasks in the system is Utilization^n. What is the chance of an overflow of the FIFO if it can hold 10 messages?

f. [18] <F.8> How big must the gateway be to have packet loss due to FIFO overflow less than one packet per million?

✪ F.32 [20] <F.8> The imbalance between the time of sending and receiving can cause problems in network performance. Sending too fast can cause the network to back up and increase the latency of messages, since the receivers will not be able to pull out the message fast enough. A technique called *bandwidth matching* proposes a simple solution: Slow down the sender so that it matches the performance of the receiver [Brewer and Kuszmaul 1994]. If two machines exchange an equal number of messages using a protocol like UDP, one will get ahead of the other, causing it to send all its messages first. After the receiver puts all these messages away, it will then send its messages. Estimate the performance for this case versus a bandwidth-matched case. Assume that the send overhead is 200 μs, the receive overhead is 300 μs, time of flight is 5 μs, latency is 10 μs, and that the two machines want to exchange 100 messages.

F.33 [40] <F.8> Compare the performance of UDP with and without bandwidth matching by slowing down the UDP send code to match the receive code as advised by bandwidth matching [Brewer and Kuszmaul 1994]. Devise an experiment to see how much performance changes as a result. How should you change the send rate when two nodes send to the same destination? What if one sender sends to two destinations?

✪ F.34 [40] <F.6, F.8> If you have access to an SMP and a cluster, write a program to measure latency of communication and bandwidth of communication between processors, as was plotted in Figure F.32 on page F-80.

F.35 [20/20/20] <F.9> If you have access to a UNIX system, use `ping` to explore the Internet. First read the manual page. Then use `ping` without option flags to be sure you can reach the following sites. It should say that `X is alive`. Depending on your system, you may be able to see the path by setting the flags to verbose mode

(`-v`) and trace route mode (`-R`) to see the path between your machine and the example machine. Alternatively, you may need to use the program `traceroute` to see the path. If so, try its manual page. You may want to use the UNIX command `script` to make a record of your session.

a. [20] < F.9 > Trace the route to another machine on the same local area network. What is the latency?

b. [20] < F.9 > Trace the route to another machine on your campus that is *not* on the same local area network. What is the latency?

c. [20] < F.9 > Trace the route to another machine *off campus*. For example, if you have a friend you send email to, try tracing that route. See if you can discover what types of networks are used along that route. What is the latency?

F.36 [15] < F.9 > Use FTP to transfer a file from a remote site and then between local sites on the same LAN. What is the difference in bandwidth for each transfer? Try the transfer at different times of day or days of the week. Is the WAN or LAN the bottleneck?

✪ F.37 [10/10] < F.9, F.11 > Figure F.41 on page F-93 compares latencies for a high-bandwidth network with high overhead and a low-bandwidth network with low overhead for different TCP/IP message sizes.

a. [10] < F.9, F.11 > For what message sizes is the delivered bandwidth higher for the high-bandwidth network?

b. [10] < F.9, F.11 > For your answer to part (a), what is the delivered bandwidth for each network?

✪ F.38 [15] < F.9, F.11 > Using the statistics in Figure F.41 on page F-93, estimate the per-message overhead for each network.

✪ F.39 [15] < F.9, F.11 > Exercise F.37 calculates which message sizes are faster for two networks with different overhead and peak bandwidth. Using the statistics in Figure F.41 on page F-93, what is the percentage of messages that are transmitted more quickly on the network with low overhead and bandwidth? What is the percentage of data transmitted more quickly on the network with high overhead and bandwidth?

✪ F.40 [15] < F.9, F.11 > One interesting measure of the latency and bandwidth of an inter-connection is to calculate the size of a message needed to achieve one-half of the peak bandwidth. This halfway point is sometimes referred to as $n_{1/2}$, taken from the terminology of vector processing. Using Figure F.41 on page F-93, estimate $n_{1/2}$ for TCP/IP message using 155-Mbit/sec ATM and 10-Mbit/sec Ethernet.

F.41 [Discussion] < F.10 > The Google cluster used to be constructed from 1 rack unit (RU) PCs, each with one processor and two disks. Today there are considerably denser options. How much less floor space would it take if we were to replace the 1 RU PCs with modern alternatives? Go to the Compaq or Dell Web sites to find the densest alternative. What would be the estimated impact on cost of the equipment? What would be the estimated impact on rental cost of floor space?

What would be the impact on interconnection network design for achieving power/performance efficiency?

F.42 [Discussion] < F.13 > At the time of the writing of the fourth edition, it was unclear what would happen with Ethernet versus InfiniBand versus Advanced Switching in the machine room. What are the technical advantages of each? What are the economic advantages of each? Why would people maintaining the system prefer one to the other? How popular is each network today? How do they compare to proprietary commercial networks such as Myrinet and Quadrics?

G.1 Introduction G-2
G.2 Vector Performance in More Depth G-2
G.3 Vector Memory Systems in More Depth G-9
G.4 Enhancing Vector Performance G-11
G.5 Effectiveness of Compiler Vectorization G-14
G.6 Putting It All Together: Performance of Vector Processors G-15
G.7 A Modern Vector Supercomputer: The Cray X1 G-21
G.8 Concluding Remarks G-25
G.9 Historical Perspective and References G-26
Exercises G-29

G

Vector Processors in More Depth

Revised by Krste Asanovic
Massachusetts Institute of Technology

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors.…One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

Seymour Cray
Public lecture at Lawrence Livermore Laboratorieson on the introduction of the Cray-1 (1976)

G.1 Introduction

Chapter 4 introduces vector architectures and places Multimedia SIMD extensions and GPUs in proper context to vector architectures.

In this appendix, we go into more detail on vector architectures, including more accurate performance models and descriptions of previous vector architectures. Figure G.1 shows the characteristics of some typical vector processors, including the size and count of the registers, the number and types of functional units, the number of load-store units, and the number of lanes.

G.2 Vector Performance in More Depth

The chime approximation is reasonably accurate for long vectors. Another source of overhead is far more significant than the issue limitation.

The most important source of overhead ignored by the chime model is vector *start-up time*. The start-up time comes from the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used. The start-up time increases the effective time to execute a convoy to more than one chime. Because of our assumption that convoys do not overlap in time, the start-up time delays the execution of subsequent convoys. Of course, the instructions in successive convoys either have structural conflicts for some functional unit or are data dependent, so the assumption of no overlap is reasonable. The actual time to complete a convoy is determined by the sum of the vector length and the start-up time. If vector lengths were infinite, this start-up overhead would be amortized, but finite vector lengths expose it, as the following example shows.

Example Assume that the start-up overhead for functional units is shown in Figure G.2.

Show the time that each convoy can begin and the total number of cycles needed. How does the time compare to the chime approximation for a vector of length 64?

Answer Figure G.3 provides the answer in convoys, assuming that the vector length is *n*. One tricky question is when we assume the vector sequence is done; this determines whether the start-up time of the `SV` is visible or not. We assume that the instructions following cannot fit in the same convoy, and we have already assumed that convoys do not overlap. Thus, the total time is given by the time until the last vector instruction in the last convoy completes. This is an approximation, and the start-up time of the last vector instruction may be seen in some sequences and not in others. For simplicity, we always include it.

The time per result for a vector of length 64 is $4+(42/64)=4.65$ clock cycles, while the chime approximation would be 4. The execution time with startup overhead is 1.16 times higher.

Processor (year)	Vector clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983) Cray Y-MP (1988)	118 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/ VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/ logical, shift	1	4
Cray C90 (1991) Cray T90 (1995)	240 460	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998) SV1ex (2001)	300 500	8	64 (MSP)	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/ shift, logical	1 load-store	1
NEC SX/6 (2001)	500	8 + 64	256	4: FP or integer add/shift, multiply, divide, logical	1	8
NEC SX/8 (2004)	2000	8 + 64	256	4: FP or integer add/shift, multiply, divide, logical	1	4
Cray X1 (2002)	800	32	64 256 (MSP)	3: FP or integer, add/logical, multiply/shift, divide/square root/logical	1 load 1 store	2 8 (MSP)
Cray XIE (2005)	1130					

Figure G.1 Characteristics of several vector-register architectures. If the machine is a multiprocessor, the entries correspond to the characteristics of one processor. Several of the machines have different clock rates in the vector and scalar units; the clock rates shown are for the vector units. The Fujitsu machines' vector registers are configurable: The size and count of the 8K 64-bit entries may be varied inversely to one another (e.g., on the VP200, from eight registers each 1K elements long to 256 registers each 32 elements long). The NEC machines have eight foreground vector registers connected to the arithmetic units plus 32 to 64 background vector registers connected between the memory system and the foreground vector registers. Add pipelines perform add and subtract. The multiply/divide-add unit on the Hitachi S810/820 performs an FP multiply or divide followed by an add or subtract (while the multiply-add unit performs a multiply followed by an add or subtract). Note that most processors use the vector FP multiply and divide units for vector integer multiply and divide, and several of the processors use the same units for FP scalar and FP vector operations. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The number of lanes is the number of parallel pipelines in each of the functional units as described in Section G.4. For example, the NEC SX/5 can complete 16 multiplies per cycle in the multiply functional unit. Several machines can split a 64-bit lane into two 32-bit lanes to increase performance for applications that require only reduced precision. The Cray SV1 and Cray X1 can group four CPUs with two lanes each to act in unison as a single larger CPU with eight lanes, which Cray calls a Multi-Streaming Processor (MSP).

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply unit	7
Add unit	6

Figure G.2 Start-up overhead.

Convoy	Starting time	First-result time	Last-result time
1. `LV`	0	12	$11+n$
2. `MULVS.D LV`	$12+n$	$12+n+12$	$23+2n$
3. `ADDV.D`	$24+2n$	$24+2n+6$	$29+3n$
4. `SV`	$30+3n$	$30+3n+12$	$41+4n$

Figure G.3 Starting times and first- and last-result times for convoys 1 through 4. The vector length is *n*.

For simplicity, we will use the chime approximation for running time, incorporating start-up time effects only when we want performance that is more detailed or to illustrate the benefits of some enhancement. For long vectors, a typical situation, the overhead effect is not that large. Later in the appendix, we will explore ways to reduce start-up overhead.

Start-up time for an instruction comes from the pipeline depth for the functional unit implementing that instruction. If the initiation rate is to be kept at 1 clock cycle per result, then

$$\text{Pipeline depth} = \left\lceil \frac{\text{Total functional unit time}}{\text{Clock cycle time}} \right\rceil$$

For example, if an operation takes 10 clock cycles, it must be pipelined 10 deep to achieve an initiation rate of one per clock cycle. Pipeline depth, then, is determined by the complexity of the operation and the clock cycle time of the processor. The pipeline depths of functional units vary widely—2 to 20 stages are common—although the most heavily used units have pipeline depths of 4 to 8 clock cycles.

For VMIPS, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for loads. All functional units are fully pipelined. From Chapter 4, pipeline depths are 6 clock cycles for floating-point add and 7 clock cycles for floating-point multiply. On VMIPS, as on most vector processors, independent vector operations using different functional units can issue in the same convoy.

In addition to the start-up overhead, we need to account for the overhead of executing the strip-mined loop. This strip-mining overhead, which arises from

Operation	Start-up penalty
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figure G.4 Start-up penalties on VMIPS. These are the start-up penalties in clock cycles for VMIPS vector operations.

the need to reinitiate the vector sequence and set the Vector Length Register (VLR) effectively adds to the vector start-up time, assuming that a convoy does not overlap with other instructions. If that overhead for a convoy is 10 cycles, then the effective overhead per 64 elements increases by 10 cycles, or 0.15 cycles per element.

Two key factors contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. The number of convoys in the loop, which determines the number of chimes. We use the notation T_{chime} for the execution time in chimes.
2. The overhead for each strip-mined sequence of convoys. This overhead consists of the cost of executing the scalar code for strip-mining each block, T_{loop}, plus the vector start-up cost for each convoy, T_{start}.

There may also be a fixed overhead associated with setting up the vector sequence the first time. In recent vector processors, this overhead has become quite small, so we ignore it.

The components can be used to state the total running time for a vector sequence operating on a vector of length n, which we will call T_n:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times \left(T_{loop} + T_{start}\right) + n \times T_{chime}$$

The values of T_{start}, T_{loop}, and T_{chime} are compiler and processor dependent. The register allocation and scheduling of the instructions affect both what goes in a convoy and the start-up overhead of each convoy.

For simplicity, we will use a constant value for T_{loop} on VMIPS. Based on a variety of measurements of Cray-1 vector execution, the value chosen is 15 for T_{loop}. At first glance, you might think that this value is too small. The overhead in each loop requires setting up the vector starting addresses and the strides, incrementing counters, and executing a loop branch. In practice, these scalar instructions can be totally or partially overlapped with the vector instructions, minimizing the time spent on these overhead functions. The value of T_{loop} of course depends on the loop structure, but the dependence is slight compared with the connection between the vector code and the values of T_{chime} and T_{start}.

Example What is the execution time on VMIPS for the vector operation $A = B \times s$, where s is a scalar and the length of the vectors A and B is 200?

Answer Assume that the addresses of A and B are initially in `Ra` and `Rb`, `s` is in `Fs`, and recall that for MIPS (and VMIPS) R0 always holds 0. Since (200 mod 64) = 8, the first iteration of the strip-mined loop will execute for a vector length of 8 elements, and the following iterations will execute for a vector length of 64 elements. The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either 8 or 64, we increment the address registers by 8 × 8 = 64 after the first segment and 8 × 64 = 512 for later segments. The total number of bytes in the vector is 8 × 200 = 1600, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600. Here is the actual code:

```
      DADDUI  R2,R0,#1600 ;total # bytes in vector
      DADDU   R2,R2,Ra    ;address of the end of A vector
      DADDUI  R1,R0,#8    ;loads length of 1st segment
      MTC1    VLR,R1      ;load vector length in VLR
      DADDUI  R1,R0,#64   ;length in bytes of 1st segment
      DADDUI  R3,R0,#64   ;vector length of other segments
Loop: LV      V1,Rb       ;load B
      MULVS.D V2,V1,Fs    ;vector * scalar
      SV      Ra,V2       ;store A
      DADDU   Ra,Ra,R1    ;address of next segment of A
      DADDU   Rb,Rb,R1    ;address of next segment of B
      DADDUI  R1,R0,#512  ;load byte offset next segment
      MTC1    VLR,R3      ;set length to 64 elements
      DSUBU   R4,R2,Ra    ;at the end of A?
      BNEZ    R4,Loop     ;if not, go back
```

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{\text{chime}} = 3$. Let's use our basic formula:

$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

$$T_{200} = 4 \times (15 + T_{\text{start}}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{\text{start}}) + 600 = 660 + (4 \times T_{\text{start}})$$

The value of T_{start} is the sum of:

- The vector load start-up of 12 clock cycles
- A 7-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store

Thus, the value of T_{start} is given by:

$$T_{\text{start}} = 12 + 7 + 12 = 31$$

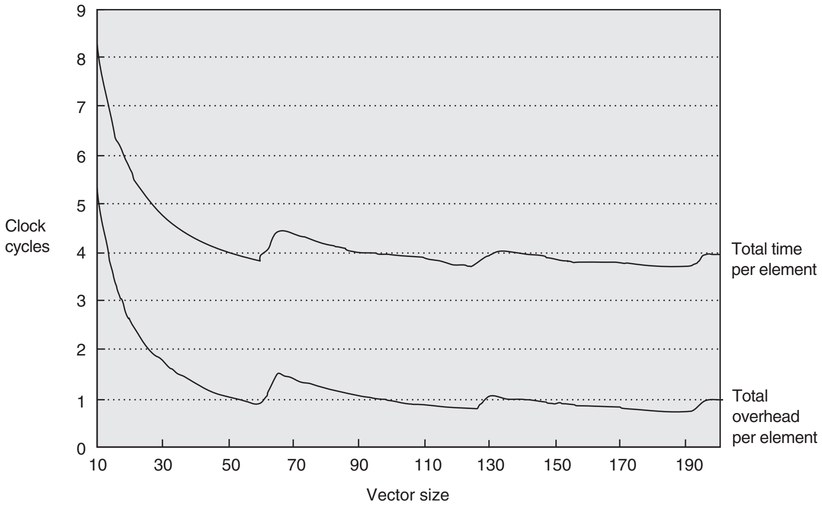


Figure G.5 The total execution time per element and the total overhead time per element versus the vector length for the example on page F-6. For short vectors, the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

So, the overall value becomes:

$$T_{200} = 660 + 4 \times 31 = 784$$

The execution time per element with all start-up costs is then 784/200 = 3.9, compared with a chime approximation of three. In Section G.4, we will be more ambitious—allowing overlapping of separate convoys.

Figure G.5 shows the overhead and effective rates per element for the previous example ($A = B \times s$) with various vector lengths. A chime-counting model would lead to 3 clock cycles per element, while the two sources of overhead add 0.9 clock cycles per element in the limit.

Pipelined Instruction Start-Up and Multiple Lanes

Adding multiple lanes increases peak performance but does not change start-up latency, and so it becomes critical to reduce start-up overhead by allowing the start of one vector instruction to be overlapped with the completion of preceding vector instructions. The simplest case to consider is when two vector instructions access a different set of vector registers. For example, in the code sequence

```
ADDV.D V1,V2,V3
ADDV.D V4,V5,V6
```

An implementation can allow the first element of the second vector instruction to follow immediately the last element of the first vector instruction down the FP adder pipeline. To reduce the complexity of control logic, some vector machines require some *recovery time* or *dead time* in between two vector instructions dispatched to the same vector unit. Figure G.6 is a pipeline diagram that shows both start-up latency and dead time for a single vector pipeline.

The following example illustrates the impact of this dead time on achievable vector performance.

Example The Cray C90 has two lanes but requires 4 clock cycles of dead time between any two vector instructions to the same functional unit, even if they have no data dependences. For the maximum vector length of 128 elements, what is the reduction in achievable peak performance caused by the dead time? What would be the reduction if the number of lanes were increased to 16?

Answer A maximum length vector of 128 elements is divided over the two lanes and occupies a vector functional unit for 64 clock cycles. The dead time adds another 4 cycles of occupancy, reducing the peak performance to $64/(64+4)=94.1\%$ of the value without dead time. If the number of lanes is increased to 16, maximum length vector instructions will occupy a functional unit for only $128/16=8$ cycles, and the dead time will reduce peak performance to $8/(8+4)=66.6\%$ of the value without dead time. In this second case, the vector units can never be more than 2/3 busy!

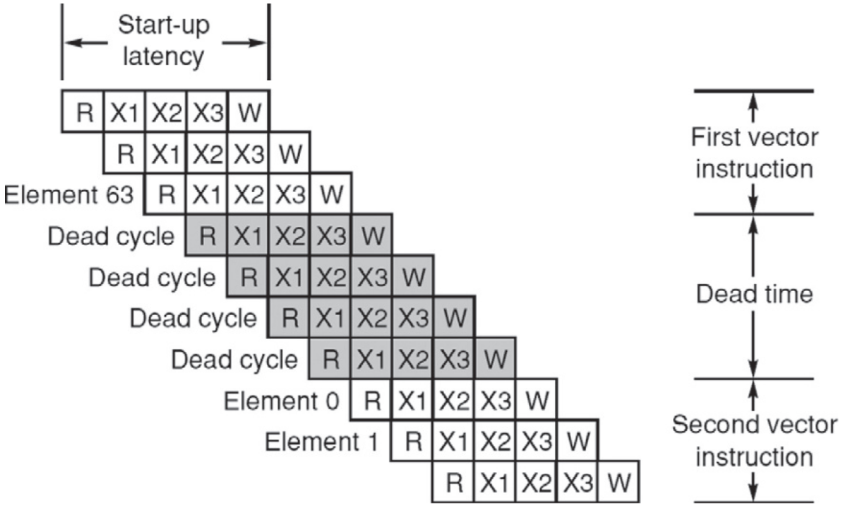


Figure G.6 Start-up latency and dead time for a single vector pipeline. Each element has a 5-cycle latency: 1 cycle to read the vector-register file, 3 cycles in execution, then 1 cycle to write the vector-register file. Elements from the same vector instruction can follow each other down the pipeline, but this machine inserts 4 cycles of dead time between two different vector instructions. The dead time can be eliminated with more complex control logic. (Reproduced with permission from Asanovic [1998].)

Pipelining instruction start-up becomes more complicated when multiple instructions can be reading and writing the same vector register and when some instructions may stall unpredictably—for example, a vector load encountering memory bank conflicts. However, as both the number of lanes and pipeline latencies increase, it becomes increasingly important to allow fully pipelined instruction start-up.

G.3 Vector Memory Systems in More Depth

To maintain an initiation rate of one word fetched or stored per clock, the memory system must be capable of producing or accepting this much data. As we saw in Chapter 4, this usually done by spreading accesses across multiple independent memory banks. Having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

The desired access rate and the bank access time determined how many banks were needed to access memory without stalls. This example shows how these timings work out in a vector processor.

Example Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes 6 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

Answer Six clocks per access require at least 6 banks, but because we want the number of banks to be a power of 2, we choose to have 8 banks. Figure G.7 shows the timing for the first few sets of accesses for an 8-bank system with a 6-clock-cycle access latency.

The timing of real memory banks is usually split into two different components, the access latency and the bank cycle time (or *bank busy time*). The access latency is the time from when the address arrives at the bank until the bank returns a data value, while the busy time is the time the bank is occupied with one request. The access latency adds to the start-up cost of fetching a vector from memory (the total memory latency also includes time to traverse the pipelined interconnection networks that transfer addresses and data between the CPU and memory banks). The bank busy time governs the effective bandwidth of a memory system because a processor cannot issue a second request to the same bank until the bank busy time has elapsed.

For simple unpipelined SRAM banks as used in the previous examples, the access latency and busy time are approximately the same. For a pipelined SRAM bank, however, the access latency is larger than the busy time because each element access only occupies one stage in the memory bank pipeline. For a DRAM bank, the access latency is usually shorter than the busy time because a DRAM needs extra time to restore the read value after the destructive read operation. For memory systems that support multiple simultaneous vector accesses

	Bank							
Cycle no.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0		136						
1		Busy	144					
2		Busy	Busy	152				
3		Busy	Busy	Busy	160			
4		Busy	Busy	Busy	Busy	168		
5		Busy	Busy	Busy	Busy	Busy	176	
6			Busy	Busy	Busy	Busy	Busy	184
7	192			Busy	Busy	Busy	Busy	Busy
8	Busy	200			Busy	Busy	Busy	Busy
9	Busy	Busy	208			Busy	Busy	Busy
10	Busy	Busy	Busy	216			Busy	Busy
11	Busy	Busy	Busy	Busy	224			Busy
12	Busy	Busy	Busy	Busy	Busy	232		
13		Busy	Busy	Busy	Busy	Busy	240	
14			Busy	Busy	Busy	Busy	Busy	248
15	256			Busy	Busy	Busy	Busy	Busy
16	Busy	264			Busy	Busy	Busy	Busy

Figure G.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all 8 banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

or allow nonsequential accesses in vector loads or stores, the number of memory banks should be larger than the minimum; otherwise, memory bank conflicts will exist.

Memory bank conflicts will not occur within a single vector memory instruction if the stride and number of banks are relatively prime with respect to each other and there are enough banks to avoid conflicts in the unit stride case. When there are no bank conflicts, multiword and unit strides run at the same rates. Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides. For example, with 64 banks, a stride of 32 will stall on every other access, rather than every access. If we originally had a stride of 8 and 16 banks, every other access would stall; with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines and/or multiple processors sharing the same memory system, we will also need more banks to prevent conflicts. Even machines with a single memory pipeline can experience memory bank conflicts on unit stride accesses between the last few elements of one instruction and the first few elements of the next instruction, and increasing the number of banks will reduce the probability of these inter-instruction conflicts. In 2011, most vector supercomputers spread the accesses from each CPU across hundreds of memory banks. Because bank conflicts can still occur in non-unit stride cases, programmers favor unit stride accesses whenever possible.

A modern supercomputer may have dozens of CPUs, each with multiple memory pipelines connected to thousands of memory banks. It would be impractical to provide a dedicated path between each memory pipeline and each memory bank, so, typically, a multistage switching network is used to connect memory pipelines to memory banks. Congestion can arise in this switching network as different vector accesses contend for the same circuit paths, causing additional stalls in the memory system.

G.4 Enhancing Vector Performance

In this section, we present techniques for improving the performance of a vector processor in more depth than we did in Chapter 4.

Chaining in More Depth

Early implementations of chaining worked like forwarding, but this restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, assuming that no structural hazard is generated. Flexible chaining requires simultaneous access to the same vector register by different vector instructions, which can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system. We assume this type of chaining throughout the rest of this appendix.

Even though a pair of operations depends on one another, chaining allows the operations to proceed in parallel on separate elements of the vector. This permits the operations to be scheduled in the same convoy and reduces the number of chimes required. For the previous sequence, a sustained rate (ignoring start-up) of two floating-point operations per clock cycle, or one chime, can be achieved, even though the operations are dependent! The total running time for the above sequence becomes:

$$\text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{Start-up time}_{\text{MULV}}$$

Figure G.8 shows the timing of a chained and an unchained version of the above pair of vector instructions with a vector length of 64. This convoy requires one chime; however, because it uses chaining, the start-up overhead will be seen in the actual timing of the convoy. In Figure G.8, the total time for chained operation is 77 clock cycles, or 1.2 cycles per result. With 128 floating-point operations done in that time, 1.7 FLOPS per clock cycle are obtained. For the unchained version, there are 141 clock cycles, or 0.9 FLOPS per clock cycle.

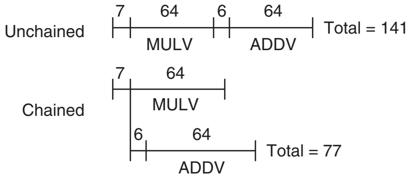


Figure G.8 Timings for a sequence of dependent vector operations `ADDV` and `MULV`, both unchained and chained. The 6- and 7-clock-cycle delays are the latency of the adder and multiplier.

Although chaining allows us to reduce the chime component of the execution time by putting two dependent instructions in the same convoy, it does not eliminate the start-up overhead. If we want an accurate running time estimate, we must count the start-up time both within and across convoys. With chaining, the number of chimes for a sequence is determined by the number of different vector functional units available in the processor and the number required by the application. In particular, no convoy can contain a structural hazard. This means, for example, that a sequence containing two vector memory instructions must take at least two convoys, and hence two chimes, on a processor like VMIPS with only one vector load-store unit.

Chaining is so important that every modern vector processor supports flexible chaining.

Sparse Matrices in More Depth

Chapter 4 shows techniques to allow programs with sparse matrices to execute in vector mode. Let's start with a quick review. In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
          do     100 i = 1,n
100              A(K(i)) = A(K(i)) + C(M(i))
```

This code implements a sparse vector sum on the arrays `A` and `C`, using index vectors `K` and `M` to designate the nonzero elements of `A` and `C`. (`A` and `C` must have the same number of nonzero elements—`n` of them.) Another common representation for sparse matrices uses a bit vector to show which elements exist and a dense vector for the nonzero elements. Often both representations exist in the same program. Sparse matrices are found in many codes, and there are many ways to implement them, depending on the data structure used in the program.

A simple vectorizing compiler could not automatically vectorize the source code above because the compiler would not know that the elements of `K` are distinct values and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it could run the loop in vector mode.

More sophisticated vectorizing compilers can vectorize the loop automatically without programmer annotations by inserting run time checks for data dependences. These run time checks are implemented with a vectorized software version of the advanced load address table (ALAT) hardware described in Appendix H for the Itanium processor. The associative ALAT hardware is replaced with a software hash table that detects if two element accesses within the same stripmine iteration are to the same address. If no dependences are detected, the stripmine iteration can complete using the maximum vector length. If a dependence is detected, the vector length is reset to a smaller value that avoids all dependency violations, leaving the remaining elements to be handled on the next iteration of the stripmined loop. Although this scheme adds considerable software overhead to the loop, the overhead is mostly vectorized for the common case where there are no dependences; as a result, the loop still runs considerably faster than scalar code (although much slower than if a programmer directive was provided).

A scatter-gather capability is included on many of the recent supercomputers. These operations often run more slowly than strided accesses because they are more complex to implement and are more susceptible to bank conflicts, but they are still much faster than the alternative, which may be a scalar loop. If the sparsity properties of a matrix change, a new index vector must be computed. Many processors provide support for computing the index vector quickly. The `CVI` (create vector index) instruction in VMIPS creates an index vector given a stride (m), where the values in the index vector are 0, m, $2 \times m$, …, $63 \times m$. Some processors provide an instruction to create a compressed index vector whose entries correspond to the positions with a one in the mask register. Other vector architectures provide a method to compress a vector. In VMIPS, we define the `CVI` instruction to always create a compressed index vector using the vector mask. When the vector mask is all ones, a standard index vector will be created.

The indexed loads-stores and the `CVI` instruction provide an alternative method to support conditional vector execution. Let us first recall code from Chapter 4:

```
         low = 1
         VL = (n mod MVL) /*find the odd-size piece*/
         do 1 j = 0,(n/MVL) /*outer loop*/
             do 10 i = low, low + VL - 1 /*runs for length VL*/
                 Y(i) = a * X(i) + Y(i) /*main operation*/
10           continue
             low = low + VL /*start of next vector*/
             VL = MVL /*reset the length to max*/
1        continue
```

Here is a vector sequence that implements that loop using CVI:

```
LV         V1,Ra          ;load vector A into V1
L.D        F0,#0          ;load FP zero into F0
SNEVS.D    V1,F0          ;sets the VM to 1 if V1(i)!=F0
CVI        V2,#8          ;generates indices in V2
POP        R1,VM          ;find the number of 1's in VM
MTC1       VLR,R1         ;load vector-length register
CVM                       ;clears the mask
```

```
LVI      V3,(Ra+V2)    ;load the nonzero A elements
LVI      V4,(Rb+V2)    ;load corresponding B elements
SUBV.D   V3,V3,V4      ;do the subtract
SVI      (Ra+V2),V3    ;store A back
```

Whether the implementation using scatter-gather is better than the conditionally executed version depends on the frequency with which the condition holds and the cost of the operations. Ignoring chaining, the running time of the original version is $5n + c_1$. The running time of the second version, using indexed loads and stores with a running time of one element per clock, is $4n + 4fn + c_2$, where f is the fraction of elements for which the condition is true (i.e., A(i) ¦ 0). If we assume that the values of c_1 and c_2 are comparable, or that they are much smaller than n, we can find when this second technique is better.

$$\text{Time}_1 = 5(n)$$

$$\text{Time}_2 = 4n + 4fn$$

We want $\text{Time}_1 > \text{Time}_2$, so

$$5n > 4n + 4fn$$
$$\frac{1}{4} > f$$

That is, the second method is faster if less than one-quarter of the elements are nonzero. In many cases, the frequency of execution is much lower. If the index vector can be reused, or if the number of vector statements within the if statement grows, the advantage of the scatter-gather approach will increase sharply.

G.5 Effectiveness of Compiler Vectorization

Two factors affect the success with which a program can be run in vector mode. The first factor is the structure of the program itself: Do the loops have true data dependences, or can they be restructured so as not to have such dependences? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded. The second factor is the capability of the compiler. While no compiler can vectorize a loop where no parallelism among the loop iterations exists, there is tremendous variation in the ability of compilers to determine whether a loop can be vectorized. The techniques used to vectorize programs are the same as those discussed in Chapter 3 for uncovering ILP; here, we simply review how well these techniques work.

There is tremendous variation in how well different compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure G.9, which shows the extent of vectorization for different processors using a test suite of 100 handwritten FORTRAN kernels. The kernels were designed to test vectorization capability and can all be vectorized by hand; we will see several examples of these loops in the exercises.

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure G.9 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

G.6 Putting It All Together: Performance of Vector Processors

In this section, we look at performance measures for vector processors and what they tell us about the processors. To determine the performance of a processor on a vector problem we must look at the start-up cost and the sustained rate. The simplest and best way to report the performance of a vector processor on a loop is to give the execution time of the vector loop. For vector loops, people often give the MFLOPS (millions of floating-point operations per second) rating rather than execution time. We use the notation R_n for the MFLOPS rating on a vector of length n. Using the measurements T_n (time) or R_n (rate) is equivalent if the number of FLOPS is agreed upon. In any event, either measurement should include the overhead.

In this section, we examine the performance of VMIPS on a DAXPY loop (see Chapter 4) by looking at performance from different viewpoints. We will continue to compute the execution time of a vector loop using the equation developed in Section G.2. At the same time, we will look at different ways to measure performance using the computed time. The constant values for T_{loop} used in this section introduce some small amount of error, which will be ignored.

Measures of Vector Performance

Because vector length is so important in establishing the performance of a processor, length-related measures are often applied in addition to time and MFLOPS. These length-related measures tend to vary dramatically across different processors and are interesting to compare. (Remember, though, that *time* is always the measure of interest when comparing the relative speed of two processors.) Three of the most important length-related measures are

- R_∞—The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems have limited vector lengths, and the overhead penalties encountered in real problems will be larger.
- $N_{1/2}$—The vector length needed to reach one-half of R_∞. This is a good measure of the impact of overhead.
- N_v—The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

Let's look at these measures for our DAXPY problem running on VMIPS. When chained, the inner loop of the DAXPY code in convoys looks like Figure G.10 (assuming that `Rx` and `Ry` hold starting addresses).

Recall our performance equation for the execution time of a vector loop with n elements, T_n:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Chaining allows the loop to run in three chimes (and no less, since there is one memory pipeline); thus, $T_{chime} = 3$. If T_{chime} were a complete indication of performance, the loop would run at an MFLOPS rate of 2/3 × clock rate (since there are 2 FLOPS per iteration). Thus, based only on the chime count, a 500 MHz VMIPS would run this loop at 333 MFLOPS assuming no strip-mining or start-up overhead. There are several ways to improve the performance: Add additional vector load-store units, allow convoys to overlap to reduce the impact of start-up overheads, and decrease the number of loads required by vector-register allocation. We will examine the first two extensions in this section. The last optimization is actually used for the Cray-1, VMIPS's cousin, to boost the performance by 50%. Reducing the number of loads requires an interprocedural optimization; we examine this transformation in Exercise G.6. Before we examine the first two extensions, let's see what the real performance, including overhead, is.

`LV V1,Rx`	`MULVS.D V2,V1,F0`	Convoy 1: chained load and multiply
`LV V3,Ry`	`ADDV.D V4,V2,V3`	Convoy 2: second load and add, chained
`SV Ry,V4`		Convoy 3: store the result

Figure G.10 The inner loop of the DAXPY code in chained convoys.

The Peak Performance of VMIPS on DAXPY

First, we should determine what the peak performance, R_∞, really is, since we know it must differ from the ideal 333 MFLOPS rate. For now, we continue to use the simplifying assumption that a convoy cannot start until all the instructions in an earlier convoy have completed; later we will remove this restriction. Using this simplification, the start-up overhead for the vector sequence is simply the sum of the start-up times of the instructions:

$$T_{\text{start}} = 12 + 7 + 12 + 6 + 12 = 49$$

Using $\text{MVL} = 64$, $T_{\text{loop}} = 15$, $T_{\text{start}} = 49$, and $T_{\text{chime}} = 3$ in the performance equation, and assuming that n is not an exact multiple of 64, the time for an n-element operation is

$$\begin{aligned} T_n &= \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n \\ &\le (n + 64) + 3n \\ &= 4n + 64 \end{aligned}$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute R_∞ for a 500 MHz clock as:

$$R_\infty = \lim_{n\to\infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of n, hence

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n\to\infty} (\text{Clock cycles per iteration})}$$

$$\lim_{n\to\infty} (\text{Clock cycles per iteration}) = \lim_{n\to\infty} \left(\frac{T_n}{n} \right) = \lim_{n\to\infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$

The performance without the start-up overhead, which is the peak performance given the vector functional unit structure, is now 1.33 times higher. In actuality, the gap between peak and sustained performance for this benchmark is even larger!

Sustained Performance of VMIPS on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a 100 × 100 matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length k is used k times. Thus, the average vector length is given by:

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66:

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 500}{326}\ \text{MFLOPS} = 202\ \text{MFLOPS}$$

The peak number, ignoring start-up overhead, is 1.64 times higher than this estimate of sustained performance on the real vector lengths. In actual practice, the Linpack benchmark contains a nontrivial fraction of code that cannot be vectorized. Although this code accounts for less than 20% of the time before vectorization, it runs at less than one-tenth of the performance when counted as FLOPS. Thus, Amdahl's law tells us that the overall performance will be significantly lower than the performance estimated from analyzing the inner loop.

Since vector length has a significant impact on performance, the $N_{1/2}$ and N_v measures are often used in comparing vector machines.

Example What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 500$$

Simplifying this and then assuming $N_{1/2} < 64$, so that $T_{N_{1/2}<64} = 64 + 3 \times n$, yields:

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.

Example What is the vector length, N_v, such that the vector operation runs faster than the scalar?

Answer Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as $10 + 12 + 12 + 7 + 6 + 12 = 59$ clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \le 64} = 64 + 3 \times n$ clock cycles. Therefore,

$$64 + 3N_v = 59N_v$$

$$N_v = \left\lceil \frac{64}{56} \right\rceil$$

$$N_v = 2$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small.

DAXPY Performance on an Enhanced VMIPS

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory access pipelines. This is the major architectural difference between the Cray X-MP (and later processors) and the Cray-1. The Cray X-MP has three memory pipelines, compared with the Cray-1's single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

Example What would be the value of T_{66} for DAXPY on VMIPS if we added two more memory pipelines?

Answer With three memory pipelines, all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{start}) + 66 \times T_{chime}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

With three memory pipelines, we have reduced the clock cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl's law: We improved the theoretical peak rate as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance.

Another improvement could come from allowing different convoys to overlap and also allowing the scalar loop overhead to overlap with the vector instructions. This requires that one vector operation be allowed to begin using a functional unit before another operation has completed, which complicates the instruction issue logic. Allowing this overlap eliminates the separate start-up overhead for every convoy except the first and hides the loop overhead as well.

To achieve the maximum hiding of strip-mining overhead, we need to be able to overlap strip-mined instances of the loop, allowing two instances of a convoy as well as possibly two instances of the scalar code to be in execution simultaneously. This requires the same techniques we looked at in Chapter 3 to avoid WAR hazards, although because no overlapped read and write of a single vector element is possible, copying can be avoided. This technique, called *tailgating*, was used in the Cray-2. Alternatively, we could unroll the outer loop to create several instances of the vector sequence using different register sets (assuming sufficient registers), just as we did in Chapter 3. By allowing maximum overlap of the convoys and the scalar loop overhead, the start-up and loop overheads will only be seen *once* per vector sequence, independent of the number of convoys and the instructions in each convoy. In this way, a processor with vector registers can have both low start-up overhead for short vectors and high peak performance for very long vectors.

Example What would be the values of R_∞ and T_{66} for DAXPY on VMIPS if we added two more memory pipelines and allowed the strip-mining and start-up overheads to be fully overlapped?

Answer

$$R_\infty = \lim_{n\to\infty}\left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}}\right)$$

$$\lim_{n\to\infty}(\text{Clock cycles per iteration}) = \lim_{n\to\infty}\left(\frac{T_n}{n}\right)$$

Since the overhead is only seen once, $T_n = n + 49 + 15 = n + 64$. Thus,

$$\lim_{n\to\infty}\left(\frac{T_n}{n}\right) = \lim_{n\to\infty}\left(\frac{n+64}{n}\right) = 1$$

$$R_\infty = \frac{2 \times 500\ \text{MHz}}{1} = 1000\ \text{MFLOPS}$$

Adding the extra memory pipelines and more flexible issue logic yields an improvement in peak performance of a factor of 4. However, $T_{66} = 130$, so for shorter vectors the sustained performance improvement is about $326/130 = 2.5$ times.

In summary, we have examined several measures of vector performance. Theoretical peak performance can be calculated based purely on the value of T_{chime} as:

$$\frac{\text{Number of FLOPS per iteration} \times \text{Clock rate}}{T_{\text{chime}}}$$

By including the loop overhead, we can calculate values for peak performance for an infinite-length vector (R_∞) and also for sustained performance, R_n for a vector of length n, which is computed as:

$$R_n = \frac{\text{Number of FLOPS per iteration} \times n \times \text{Clock rate}}{T_n}$$

Using these measures we also can find $N_{1/2}$ and N_v, which give us another way of looking at the start-up overhead for vectors and the ratio of vector to scalar speed. A wide variety of measures of performance of vector processors is useful in understanding the range of performance that applications may see on a vector processor.

G.7 A Modern Vector Supercomputer: The Cray X1

The Cray X1 was introduced in 2002, and, together with the NEC SX/8, represents the state of the art in modern vector supercomputers. The X1 system architecture supports thousands of powerful vector processors sharing a single global memory.

The Cray X1 has an unusual processor architecture, shown in Figure G.11. A large Multi-Streaming Processor (MSP) is formed by ganging together four Single-Streaming Processors (SSPs). Each SSP is a complete single-chip vector microprocessor, containing a scalar unit, scalar caches, and a two-lane vector unit. The SSP scalar unit is a dual-issue out-of-order superscalar processor with a 16 KB instruction cache and a 16 KB scalar write-through data cache, both two-way set associative with 32-byte cache lines. The SSP vector unit contains a vector register file, three vector arithmetic units, and one vector load-store unit. It is much easier to pipeline deeply a vector functional unit than a superscalar issue mechanism, so the X1 vector unit runs at twice the clock rate (800 MHz) of the scalar unit (400 MHz). Each lane can perform a 64-bit floating-point add and a 64-bit floating-point multiply each cycle, leading to a peak performance of 12.8 GFLOPS per MSP.

All previous Cray machines could trace their instruction set architecture (ISA) lineage back to the original Cray-1 design from 1976, with 8 primary registers each for addresses, scalar data, and vector data. For the X1, the ISA was redesigned from scratch to incorporate lessons learned over the last 30 years of compiler and microarchitecture research. The X1 ISA includes 64 64-bit scalar address registers and 64 64-bit scalar data registers, with 32 vector data registers (64 bits per element) and 8 vector mask registers (1 bit per element). The large increase in the number of registers allows the compiler to map more program variables into registers to reduce memory traffic and also allows better static scheduling of code to improve

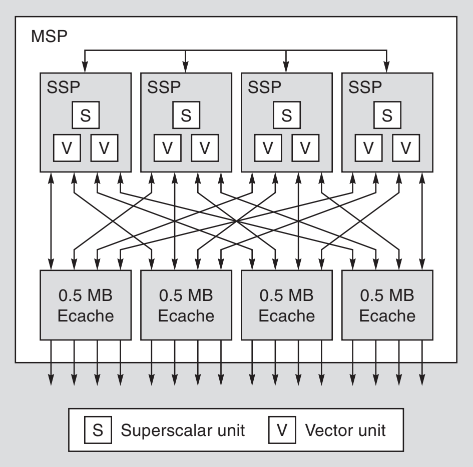


Figure G.11 Cray MSP module. (From Dunnigan et al. [2005].)

run time overlap of instruction execution. Earlier Crays had a compact variable-length instruction set, but the X1 ISA has fixedlength instructions to simplify superscalar fetch and decode.

Four SSP chips are packaged on a multichip module together with four cache chips implementing an external 2 MB cache (Ecache) shared by all the SSPs. The Ecache is two-way set associative with 32-byte lines and a write-back policy. The Ecache can be used to cache vectors, reducing memory traffic for codes that exhibit temporal locality. The ISA also provides vector load and store instruction variants that do not allocate in cache to avoid polluting the Ecache with data that is known to have low locality. The Ecache has sufficient bandwidth to supply one 64-bit word per lane per 800 MHz clock cycle, or over 50 GB/sec per MSP.

At the next level of the X1 packaging hierarchy, shown in Figure G.12, four MSPs are placed on a single printed circuit board together with 16 memory controller chips and DRAM to form an X1 node. Each memory controller chip has eight separate Rambus DRAM channels, where each channel provides 1.6 GB/sec of memory bandwidth. Across all 128 memory channels, the node has over 200 GB/sec of main memory bandwidth.

An X1 system can contain up to 1024 nodes (4096 MSPs or 16,384 SSPs), connected via a very high-bandwidth global network. The network connections are made via the memory controller chips, and all memory in the system is directly accessible from any processor using load and store instructions. This provides much faster global communication than the message-passing protocols used in cluster-based systems. Maintaining cache coherence across such a large number of high-bandwidth shared-memory nodes would be challenging. The approach taken in the X1 is to restrict each Ecache to cache data only from the local node DRAM. The memory controllers implement a directory scheme to maintain

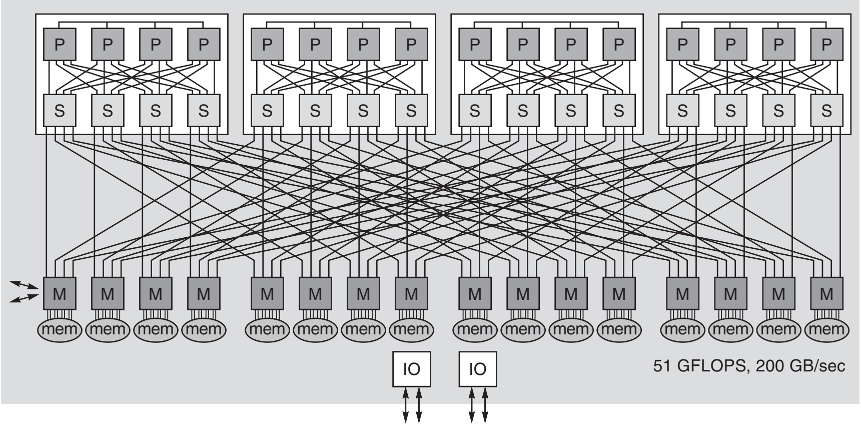


Figure G.12 Cray X1 node. (From Tanqueray [2002].)

coherency between the four Ecaches on a node. Accesses from remote nodes will obtain the most recent version of a location, and remote stores will invalidate local Ecaches before updating memory, but the remote node cannot cache these local locations.

Vector loads and stores are particularly useful in the presence of long-latency cache misses and global communications, as relatively simple vector hardware can generate and track a large number of in-flight memory requests. Contemporary superscalar microprocessors support only 8 to 16 outstanding cache misses, whereas each MSP processor can have up to 2048 outstanding memory requests (512 per SSP). To compensate, superscalar microprocessors have been moving to larger cache line sizes (128 bytes and above) to bring in more data with each cache miss, but this leads to significant wasted bandwidth on non-unit stride accesses over large datasets. The X1 design uses short 32-byte lines throughout to reduce bandwidth waste and instead relies on supporting many independent cache misses to sustain memory bandwidth. This latency tolerance together with the huge memory bandwidth for non-unit strides explains why vector machines can provide large speedups over superscalar microprocessors for certain codes.

Multi-Streaming Processors

The Multi-Streaming concept was first introduced by Cray in the SV1, but has been considerably enhanced in the X1. The four SSPs within an MSP share Ecache, and there is hardware support for barrier synchronization across the four SSPs within an MSP. Each X1 SSP has a two-lane vector unit with 32 vector registers each holding 64 elements. The compiler has several choices as to how to use the SSPs within an MSP.

The simplest use is to gang together four two-lane SSPs to emulate a single eight-lane vector processor. The X1 provides efficient barrier synchronization primitives between SSPs on a node, and the compiler is responsible for generating the MSP code. For example, for a vectorizable inner loop over 1000 elements, the compiler will allocate iterations 0–249 to SSP0, iterations 250–499 to SSP1, iterations 500–749 to SSP2, and iterations 750–999 to SSP3. Each SSP can process its loop iterations independently but must synchronize back with the other SSPs before moving to the next loop nest.

If inner loops do not have many iterations, the eight-lane MSP will have low efficiency, as each SSP will have only a few elements to process and execution time will be dominated by start-up time and synchronization overheads. Another way to use an MSP is for the compiler to parallelize across an outer loop, giving each SSP a different inner loop to process. For example, the following nested loops scale the upper triangle of a matrix by a constant:

```
/* Scale upper triangle by constant K. */
for (row = 0; row < MAX_ROWS; row++)
   for (col = row; col < MAX_COLS; col++)
      A[row][col] = A[row][col] * K;
```

Consider the case where `MAX_ROWS` and `MAX_COLS` are both 100 elements. The vector length of the inner loop steps down from 100 to 1 over the iterations of the outer loop. Even for the first inner loop, the loop length would be much less than the maximum vector length (256) of an eight-lane MSP, and the code would therefore be inefficient. Alternatively, the compiler can assign entire inner loops to a single SSP. For example, SSP0 might process rows 0, 4, 8, and so on, while SSP1 processes rows 1, 5, 9, and so on. Each SSP now sees a longer vector. In effect, this approach parallelizes the scalar overhead and makes use of the individual scalar units within each SSP.

Most application code uses MSPs, but it is also possible to compile code to use all the SSPs as individual processors where there is limited vector parallelism but significant thread-level parallelism.

Cray X1E

In 2004, Cray announced an upgrade to the original Cray X1 design. The X1E uses newer fabrication technology that allows two SSPs to be placed on a single chip, making the X1E the first multicore vector microprocessor. Each physical node now contains eight MSPs, but these are organized as two logical nodes of four MSPs each to retain the same programming model as the X1. In addition, the clock rates were raised from 400 MHz scalar and 800 MHz vector to 565 MHz scalar and 1130 MHz vector, giving an improved peak performance of 18 GFLOPS.

G.8 Concluding Remarks

During the 1980s and 1990s, rapid performance increases in pipelined scalar processors led to a dramatic closing of the gap between traditional vector supercomputers and fast, pipelined, superscalar VLSI microprocessors. In 2011, it is possible to buy a laptop computer for under $1000 that has a higher CPU clock rate than any available vector supercomputer, even those costing tens of millions of dollars. Although the vector supercomputers have lower clock rates, they support greater parallelism using multiple lanes (up to 16 in the Japanese designs) versus the limited multiple issue of the superscalar microprocessors. Nevertheless, the peak floating-point performance of the low-cost microprocessors is within a factor of two of the leading vector supercomputer CPUs. Of course, high clock rates and high peak performance do not necessarily translate into sustained application performance. Main memory bandwidth is the key distinguishing feature between vector supercomputers and superscalar microprocessor systems.

Providing this large non-unit stride memory bandwidth is one of the major expenses in a vector supercomputer, and traditionally SRAM was used as main memory to reduce the number of memory banks needed and to reduce vector start-up penalties. While SRAM has an access time several times lower than that of DRAM, it costs roughly 10 times as much per bit! To reduce main memory costs and to allow larger capacities, all modern vector supercomputers now use DRAM for main memory, taking advantage of new higher-bandwidth DRAM interfaces such as synchronous DRAM.

This adoption of DRAM for main memory (pioneered by Seymour Cray in the Cray-2) is one example of how vector supercomputers have adapted commodity technology to improve their price-performance. Another example is that vector supercomputers are now including vector data caches. Caches are not effective for all vector codes, however, so these vector caches are designed to allow high main memory bandwidth even in the presence of many cache misses. For example, the Cray X1 MSP can have 2048 outstanding memory loads; for microprocessors, 8 to 16 outstanding cache misses per CPU are more typical maximum numbers.

Another example is the demise of bipolar ECL or gallium arsenide as technologies of choice for supercomputer CPU logic. Because of the huge investment in CMOS technology made possible by the success of the desktop computer, CMOS now offers competitive transistor performance with much greater transistor density and much reduced power dissipation compared with these more exotic technologies. As a result, all leading vector supercomputers are now built with the same CMOS technology as superscalar microprocessors. The primary reason why vector supercomputers have lower clock rates than commodity microprocessors is that they are developed using standard cell ASIC techniques rather than full custom circuit design to reduce the engineering design cost. While a microprocessor design may sell tens of millions of copies and can amortize the design cost over this large number of units, a vector supercomputer is considered a success if over a hundred units are sold!

Conversely, via superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems, such as with the Multimedia SIMD extensions. As we showed in Chapter 4, the investment in hardware for SIMD performance is increasing rapidly, perhaps even more than for multiprocessors. If the even wider SIMD units of GPUs become well integrated with the scalar cores, including scatter-gather support, we may well conclude that vector architectures have won the architecture wars!

G.9 Historical Perspective and References

This historical perspective adds some details and references that were left out of the version in Chapter 4.

The CDC STAR processor and its descendant, the CYBER 205, were memory-memory vector processors. To keep the hardware simple and support the high bandwidth requirements (up to three memory references per floating-point operation), these processors did not efficiently handle non-unit stride. While most loops have unit stride, a non-unit stride loop had poor performance on these processors because memory-to-memory data movements were required to gather together (and scatter back) the nonadjacent vector elements; these operations used special scatter-gather instructions. In addition, there was special support for sparse vectors that used a bit vector to represent the zeros and nonzeros and a dense vector of nonzero values. These more complex vector operations were slow because of the long memory latency, and it was often faster to use scalar mode for sparse or non-unit stride operations. Schneck [1987] described several of the early pipelined processors (e.g., Stretch) through the first vector processors, including the 205 and Cray-1. Dongarra [1986] did another good survey, focusing on more recent processors.

The 1980s also saw the arrival of smaller-scale vector processors, called mini-supercomputers. Priced at roughly one-tenth the cost of a supercomputer ($0.5 to $1 million versus $5 to $10 million), these processors caught on quickly. Although many companies joined the market, the two companies that were most successful were Convex and Alliant. Convex started with the uniprocessor C-1 vector processor and then offered a series of small multiprocessors, ending with the C-4 announced in 1994. The keys to the success of Convex over this period were their emphasis on Cray software capability, the effectiveness of their compiler (see Figure G.9), and the quality of their UNIX OS implementation. The C-4 was the last vector machine Convex sold; they switched to making large-scale multiprocessors using Hewlett-Packard RISC microprocessors and were bought by HP in 1995. Alliant [1987] concentrated more on the multiprocessor aspects; they built an eight-processor computer, with each processor offering vector capability. Alliant ceased operation in the early 1990s.

In the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 GFLOPS. The ETA processor was delivered in the late 1980s (see Fazio [1987]) and used low-temperature CMOS in a configuration with up to 10 processors. Each processor retained the memory-memory architecture based on the CYBER 205. Although the ETA-10 achieved enormous peak performance, its scalar speed was not comparable. In 1989, CDC, the first supercomputer vendor, closed ETA and left the supercomputer design business.

In 1986, IBM introduced the System/370 vector architecture (see Moore et al. [1987]) and its first implementation in the 3090 Vector Facility. The architecture extended the System/370 architecture with 171 vector instructions. The 3090/VF was integrated into the 3090 CPU. Unlike most other vector processors of the time, the 3090/VF routed its vectors through the cache. The IBM 370 machines continued to evolve over time and are now called the IBM zSeries. The vector extensions have been removed from the architecture and some of the opcode space was reused to implement 64-bit address extensions.

In late 1989, Cray Research was split into two companies, both aimed at building high-end processors available in the early 1990s. Seymour Cray headed the spin-off, Cray Computer Corporation, until its demise in 1995. Their initial processor, the Cray-3, was to be implemented in gallium arsenide, but they were unable to develop a reliable and cost-effective implementation technology. A single Cray-3 prototype was delivered to the National Center for Atmospheric Research (NCAR) for evaluation purposes in 1993, but no paying customers were found for the design. The Cray-4 prototype, which was to have been the first processor to run at 1 GHz, was close to completion when the company filed for bankruptcy. Shortly before his tragic death in a car accident in 1996, Seymour Cray started yet another company, SRC Computers, to develop high-performance systems but this time using commodity components. In 2000, SRC announced the SRC-6 system, which combined 512 Intel microprocessors, 5 billion gates of reconfigurable logic, and a high-performance vector-style memory system.

Cray Research focused on the C90, a new high-end processor with up to 16 processors and a clock rate of 240 MHz. This processor was delivered in 1991. The J90 was a CMOS-based vector machine using DRAM memory starting at $250,000, but with typical configurations running about $1 million. In mid-1995, Cray Research was acquired by Silicon Graphics, and in 1998 released the SV1 system, which grafted considerably faster CMOS processors onto the J90 memory system, and which also added a data cache for vectors to each CPU to help meet the increased memory bandwidth demands. The SV1 also introduced the MSP concept, which was developed to provide competitive single-CPU performance by ganging together multiple slower CPUs. Silicon Graphics sold Cray Research to Tera Computer in 2000, and the joint company was renamed Cray Inc.

The basis for modern vectorizing compiler technology and the notion of data dependence was developed by Kuck and his colleagues [1974] at the University of Illinois. Banerjee [1979] developed the test named after him. Padua and Wolfe [1986] gave a good overview of vectorizing compiler technology.

Benchmark studies of various supercomputers, including attempts to understand the performance differences, have been undertaken by Lubeck, Moore, and Mendez [1985], Bucher [1983], and Jordan [1987]. There are several benchmark suites aimed at scientific usage and often employed for supercomputer benchmarking, including Linpack and the Lawrence Livermore Laboratories FORTRAN kernels. The University of Illinois coordinated the collection of a set of benchmarks for supercomputers, called the Perfect Club. In 1993, the Perfect Club was integrated into SPEC, which released a set of benchmarks, SPEChpc96, aimed at high-end scientific processing in 1996. The NAS parallel benchmarks developed at the NASA Ames Research Center [Bailey et al. 1991] have become a popular set of kernels and applications used for supercomputer evaluation. A new benchmark suite, HPC Challenge, was introduced consisting of a few kernels that stress machine memory and interconnect bandwidths in addition to floating-point performance [Luszczek et al. 2005]. Although standard supercomputer benchmarks are useful as a rough measure of machine capabilities, large supercomputer purchases are generally preceded by a careful performance evaluation on the actual mix of applications required at the customer site.

References

Alliant Computer Systems Corp, 1987. Alliant FX/Series: Product Summary. Mass, Acton (June).

Asanovic, K., 1998. Vector microprocessors," Ph.D. thesis, Computer Science Division. University of California at Berkeley (May).

Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K., 1991. The NAS parallel benchmarks. Int'l. J. Supercomputing Applications 5, 63–73.

Banerjee, U., 1979. Speedup of ordinary programs," Ph.D. thesis, Department of Computer Science. University of Illinois at Urbana-Champaign (October).

Baskett, F., Keller, T.W., 1977. An Evaluation of the Cray-1 Processor. In: Kuck, D.J., Lawrie, D.H., Sameh, A.H. (Eds.), High Speed Computer and Algorithm Organization. Academic Press, San Diego, pp. 71–84.

Brandt, M., Brooks, J., Cahir, M., Hewitt, T., Lopez-Pineda, E., Sandness, D., 2000. The Benchmarker's Guide for Cray SV1 Systems. Cray Inc., Seattle, Wash.

Bucher, I.Y., 1983. The computational speed of supercomputers. In: Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, August 29–31, 1983. Minneapolis, Minn, pp. 151–165.

Callahan, D., Dongarra, J., Levine, D., 1988. Vectorizing compilers: A test suite and results. In: Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing, November 12–17, pp. 98–105. Orlando, FL.

Chen, S., 1983. Large-scale and high-speed multiprocessor system for scientific applications. In: Hwang, K. (Ed.), Superprocessors: Design and applications. Proc. NATO Advanced Research Workshop on High-Speed Computing, June 20–22, 1983, Julich, Kernforschungsanlage, Federal Republic of Germany. IEEE, (August), 1984.

Dongarra, J.J., 1986. A survey of high performance processors. COMPCON, IEEE, 8–11 (March).

Dunnigan, T.H., Vetter, J.S., White III, J.B., Worley, P.H., 2005. Performance evaluation of the Cray X1 distributed shared-memory architecture. IEEE Micro 25 (1 (January–February)), 30–40.

Fazio, D., 1987. It's really much more fun building a supercomputer than it is simply inventing one. COMPCON, IEEE, 102–105 (February).

Flynn, M.J., 1966. Very high-speed computing systems. In: Proc. IEEE 54:12 (December), pp. 1901–1909.

Hintz, R.G., Tate, D.P., 1972. Control data STAR-100 processor design. COMPCON, IEEE 1–4 (September).

Jordan, K.E., 1987. Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers. Computer 20 (3 (March)), 10–23.

Kitagawa, K., Tagaya, S., Hagihara, Y., Kanoh, Y., 2003. A hardware overview of SX-6 and SX-7 supercomputer. NEC Research & Development J 44 (1 (January)), 2–7.

Kuck, D., Budnik, P.P., Chen, S.-C., Lawrie, D.H., Towle, R.A., Strebendt, R.E., Davis Jr., E.W., Han, J., Kraska, P.W., Muraoka, Y., 1974. Measurements of parallelism in ordinary FORTRAN programs. Computer 7 (1 (January)), 37–46.

Lincoln, N.R., 1982. Technology and design trade offs in the creation of a modern supercomputer. IEEE Trans. on Computers, 363–376. C-31:5 (May).

Lubeck, O., Moore, J., Mendez, R., 1985. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. Computer 18 (1 (January)), 10–29.

Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D., 2005. In: Introduction to the HPC challenge benchmark suite," Lawrence Berkeley National Laboratory, Paper LBNL-57493 (April 25). http://repositories.cdlib.org/lbnl/LBNL-57493.

Miranker, G.S., Rubenstein, J., Sanguinetti, J., 1988. Squeezing a Cray-class supercomputer into a single-user package. COMPCON, IEEE, 452–456 (March).

Miura, K., Uchida, K., 1983. FACOM vector processing system: VP100/200. In: Proc. NATO Advanced Research Workshop on High-Speed Computing. June 20–22, 1983, Julich, Kernforschungsanlage, Federal Republic of Germany; also in K. Hwang, ed., "Superprocessors: Design and applications," *IEEE* (August), 1984, 59–73.

Moore, B., Padegs, A., Smith, R., Bucholz, W., 1987. Concepts of the System/370 vector architecture. In: Proc. 14th Int'l. Symposium on Computer Architecture, June 3–6, 1987. Pittsburgh, Penn, pp. 282–292.

Padua, D., Wolfe, M., 1986. Advanced compiler optimizations for supercomputers. Comm. ACM 29 (12 (December)), 1184–1201.

Russell, R.M., 1978. The Cray-1 processor system. Comm. of the ACM 21 (1 (January)), 63–72.

Schneck, P.B., 1987. Superprocessor Architecture. Kluwer Academic Publishers, Norwell, Mass.

Smith, B.J., 1981. Architecture and applications of the HEP multiprocessor system. Real-Time Signal Processing IV 298, 241–248. August.

Sporer, M., Moss, F.H., Mathais, C.J., 1988. An introduction to the architecture of the Stellar Graphics supercomputer. COMPCON, IEEE 464 (March).

Tanqueray, D., 2002. The Cray X1 and supercomputer road map. In: Proc. 13th Daresbury Machine Evaluation Workshop, December 11–12. Cheshire, England.

Vajapeyam, S., 1991. Instruction-level characterization of the Cray Y-MP processor. Ph.D. thesis, Computer Sciences Department. University of Wisconsin-Madison.

Watanabe, T., 1987. Architecture and performance of the NEC supercomputer SX system. Parallel Computing 5, 247–255.

Watson, W.J., 1972. The TI ASC—a highly modular and flexible super processor architecture. In: Proc. AFIPS Fall Joint Computer Conf, pp. 221–228.

Exercises

In these exercises assume VMIPS has a clock rate of 500 MHz and that $T_{loop} = 15$. Use the start-up times from Figure G.2, and assume that the store latency is always included in the running time.

G.1 [10] < G.1, G.2 > Write a VMIPS vector sequence that achieves the peak MFLOPS performance of the processor (use the functional unit and instruction description in Section G.2). Assuming a 500-MHz clock rate, what is the peak MFLOPS?

G.2 [20/15/15] < G.1–G.6 > Consider the following vector code run on a 500 MHz version of VMIPS for a fixed vector length of 64:

```
LV       V1,Ra
MULV.D  V2,V1,V3
ADDV.D  V4,V1,V3
SV       Rb,V2
SV       Rc,V4
```

Ignore all strip-mining overhead, but assume that the store latency must be included in the time to perform the loop. The entire sequence produces 64 results.

a. [20] <G.1–G.4> Assuming no chaining and a single memory pipeline, how many chimes are required? How many clock cycles per result (including both stores as one result) does this vector sequence require, including start-up overhead?

b. [15] <G.1–G.4> If the vector sequence is chained, how many clock cycles per result does this sequence require, including overhead?

c. [15] <G.1–G.6> Suppose VMIPS had three memory pipelines and chaining. If there were no bank conflicts in the accesses for the above loop, how many clock cycles are required per result for this sequence?

G.3 [20/20/15/15/20/20/20] <G.2–G.6> Consider the following FORTRAN code:

```
          do 10 i=1,n
                 A(i)=A(i)+B(i)
                 B(i)=x * B(i)
   10     continue
```

Use the techniques of Section G.6 to estimate performance throughout this exercise, assuming a 500 MHz version of VMIPS.

a. [20] <G.2–G.6> Write the best VMIPS vector code for the inner portion of the loop. Assume `x` is in F0 and the addresses of `A` and `B` are in `Ra` and `Rb`, respectively.

b. [20] <G.2–G.6> Find the total time for this loop on VMIPS (T_{100}). What is the MFLOPS rating for the loop (R_{100})?

c. [15] <G.2–G.6> Find R_∞ for this loop.

d. [15] <G.2–G.6> Find $N_{1/2}$ for this loop.

e. [20] <G.2–G.6> Find N_v for this loop. Assume the scalar code has been pipeline scheduled so that each memory reference takes six cycles and each FP operation takes three cycles. Assume the scalar overhead is also T_{loop}.

f. [20] <G.2–G.6> Assume VMIPS has two memory pipelines. Write vector code that takes advantage of the second memory pipeline. Show the layout in convoys.

g. [20] <G.2–G.6> Compute T_{100} and R_{100} for VMIPS with two memory pipelines.

G.4 [20/10] <G.2> Suppose we have a version of VMIPS with eight memory banks (each a double word wide) and a memory access time of eight cycles.

a. [20] <G.2> If a load vector of length 64 is executed with a stride of 20 double words, how many cycles will the load take to complete?

b. [10] <G.2> What percentage of the memory bandwidth do you achieve on a 64-element load at stride 20 versus stride 1?

G.5 [12/12] < G.5–G.6 > Consider the following loop:

```
          C=0.0
          do 10 i=1,64
                A(i)=A(i)+B(i)
                C=C+A(i)
     10   continue
```

a. [12] < G.5–G.6 > Split the loop into two loops: one with no dependence and one with a dependence. Write these loops in FORTRAN—as a source-to-source transformation. This optimization is called *loop fission.*

b. [12] < G.5–G.6 > Write the VMIPS vector code for the loop without a dependence.

G.6 [20/15/20/20] < G.5–G.6 > The compiled Linpack performance of the Cray-1 (designed in 1976) was almost doubled by a better compiler in 1989. Let's look at a simple example of how this might occur. Consider the DAXPY-like loop (where *k* is a parameter to the procedure containing the loop):

```
          do 10 i=1,64
             do 10 j=1,64
             Y(k,j)=a*X(i,j)+Y(k,j)
     10   continue
```

a. [20] < G.5–G.6 > Write the *straightforward* code sequence for just the inner loop in VMIPS vector instructions.

b. [15] < G.5–G.6 > Using the techniques of Section G.6, estimate the performance of this code on VMIPS by finding T_{64} in clock cycles. You may assume that T_{loop} of overhead is incurred for each iteration of the outer loop. What limits the performance?

c. [20] < G.5–G.6 > Rewrite the VMIPS code to reduce the performance limitation; show the resulting inner loop in VMIPS vector instructions. (*Hint:* Think about what establishes T_{chime}; can you affect it?) Find the total time for the resulting sequence.

d. [20] < G.5–G.6 > Estimate the performance of your new version, using the techniques of Section G.6 and finding T_{64}.

G.7 [15/15/25] < G.4 > Consider the following code:

```
          do 10 i=1,64
             if (B(i) .ne. 0) then
                A(i)=A(i)/B(i)
     10   continue
```

Assume that the addresses of `A` and `B` are in `Ra` and `Rb`, respectively, and that F0 contains 0.

a. [15] <G.4> Write the VMIPS code for this loop using the vector-mask capability.

b. [15] <G.4> Write the VMIPS code for this loop using scatter-gather.

c. [25] <G.4> Estimate the performance (T_{100} in clock cycles) of these two vector loops, assuming a divide latency of 20 cycles. Assume that all vector instructions run at one result per clock, independent of the setting of the vector-mask register. Assume that 50% of the entries of `B` are 0. Considering hardware costs, which would you build if the above loop were typical?

G.8 [15/20/15/15] <G.1–G.6> The difference between peak and sustained performance can be large. For one problem, a Hitachi S810 had a peak speed twice as high as that of the Cray X-MP, while for another more realistic problem, the Cray X-MP was twice as fast as the Hitachi processor. Let's examine why this might occur using two versions of VMIPS and the following code sequences:

```
C            Code sequence 1
        do 10 i=1,10000
             A(i)=x * A(i)+y * A(i)
10      continue
C            Code sequence 2
        do 10 i=1,100
             A(i)=x * A(i)
10      continue
```

Assume there is a version of VMIPS (call it VMIPS-II) that has two copies of every floating-point functional unit with full chaining among them. Assume that both VMIPS and VMIPS-II have two load-store units. Because of the extra functional units and the increased complexity of assigning operations to units, all the overheads (T_{loop} and T_{start}) are doubled for VMIPS-II.

a. [15] <G.1–G.6> Find the number of clock cycles on code sequence 1 on VMIPS.

b. [20] <G.1–G.6> Find the number of clock cycles on code sequence 1 for VMIPS-II. How does this compare to VMIPS?

c. [15] <G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS.

d. [15] <G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS-II. How does this compare to VMIPS?

G.9 [20] <G.5> Here is a tricky piece of code with two-dimensional arrays. Does this loop have dependences? Can these loops be written so they are parallel? If so, how? Rewrite the *source* code so that it is clear that the loop can be vectorized, if possible.

```
        do 290 j=2,n
           do 290 i=2,j
              aa(i,j)=aa(i-1,j)*aa(i-1,j)+bb(i,j)
290      continue
```

G.10 [12/15] < G.5 > Consider the following loop:

```
        do 10 i=2,n
           A(i)=B
    10     C(i)=A(i - 1)
```

a. [12] < G.5 > Show there is a loop-carried dependence in this code fragment.

b. [15] < G.5 > Rewrite the code in FORTRAN so that it can be vectorized as two separate vector sequences.

G.11 [15/25/25] < G.5 > As we saw in Section G.5, some loop structures are not easily vectorized. One common structure is a *reduction*—a loop that reduces an array to a single value by repeated application of an operation. This is a special case of a recurrence. A common example occurs in dot product:

```
          dot=0.0
          do 10 i=1,64
    10          dot=dot+A(i) * B(i)
```

This loop has an obvious loop-carried dependence (on `dot`) and cannot be vectorized in a straightforward fashion. The first thing a good vectorizing compiler would do is split the loop to separate out the vectorizable portion and the recurrence and perhaps rewrite the loop as:

```
          do 10 i=1,64
    10           dot(i)=A(i) * B(i)
          do 20 i=2,64
    20           dot(1)=dot(1)+dot(i)
```

The variable `dot` has been expanded into a vector; this transformation is called *scalar expansion*. We can try to vectorize the second loop either relying strictly on the compiler (part (a)) or with hardware support as well (part (b)). There is an important caveat in the use of vector techniques for reduction. To make reduction work, we are relying on the associativity of the operator being used for the reduction. Because of rounding and finite range, however, floating-point arithmetic is not strictly associative. For this reason, most compilers require the programmer to indicate whether associativity can be used to more efficiently compile reductions.

a. [15] < G.5 > One simple scheme for compiling the loop with the recurrence is to add sequences of progressively shorter vectors—two 32-element vectors, then two 16-element vectors, and so on. This technique has been called *recursive doubling*. It is faster than doing all the operations in scalar mode. Show how the FORTRAN code would look for execution of the second loop in the preceding code fragment using recursive doubling.

b. [25] < G.5 > In some vector processors, the vector registers are addressable, and the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction, called *partial sums*.

The key idea in partial sums is to reduce the vector to *m* sums where *m* is the total latency through the vector functional unit, including the operand read and write times. Assume that the VMIPS vector registers are addressable (e.g., you can initiate a vector operation with the operand V1(16), indicating that the input operand began with element 16). Also, assume that the total latency for adds, including operand read and write, is eight cycles. Write a VMIPS code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

c. [25] < G.5 > Discuss how adding the extension in part (b) would affect a machine that had multiple lanes.

G.12 [40] < G.3–G.4 > Extend the MIPS simulator to be a VMIPS simulator, including the ability to count clock cycles. Write some short benchmark programs in MIPS and VMIPS assembly language. Measure the speedup on VMIPS, the percentage of vectorization, and usage of the functional units.

G.13 [50] < G.5 > Modify the MIPS compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.

G.14 [Discussion] Some proponents of vector processors might argue that the vector processors have provided the best path to ever-increasing amounts of processor power by focusing their attention on boosting peak vector performance. Others would argue that the emphasis on peak performance is misplaced because an increasing percentage of the programs are dominated by nonvector performance. (Remember Amdahl’s law?) The proponents would respond that programmers should work to make their programs vectorizable. What do you think about this argument?

H.1 Introduction: Exploiting Instruction-Level Parallelism Statically H-2
H.2 Detecting and Enhancing Loop-Level Parallelism H-2
H.3 Scheduling and Structuring Code for Parallelism H-12
H.4 Hardware Support for Exposing Parallelism: Predicated Instructions H-23
H.5 Hardware Support for Compiler Speculation H-27
H.6 The Intel IA-64 Architecture and Itanium Processor H-32
H.7 Concluding Remarks H-43

H

Hardware and Software for VLIW and EPIC

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.

M. Hopkins
in a commentary on the EPIC
approach and the IA-64 architecture (2000)

H.1 Introduction: Exploiting Instruction-Level Parallelism Statically

In this chapter, we discuss compiler technology for increasing the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques. The next section defines when a loop is parallel, how a dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependences. The following section discusses the topic of scheduling code to improve parallelism. These two sections serve as an introduction to these techniques.

We do not attempt to explain the details of ILP-oriented compiler techniques, since that would take hundreds of pages, rather than the 20 we have allotted. Instead, we view this material as providing general background that will enable the reader to have a basic understanding of the compiler techniques used to exploit ILP in modern computers.

Hardware support for these compiler techniques can greatly increase their effectiveness, and Sections H.4 and H.5 explore such support. The IA-64 represents the culmination of the compiler and hardware ideas for exploiting parallelism statically and includes support for many of the concepts proposed by researchers during more than a decade of research into the area of compiler-based instruction-level parallelism. Section H.6 provides a description and performance analyses of the Intel IA-64 architecture and its second-generation implementation, Itanium 2.

The core concepts that we exploit in statically based techniques—finding parallelism, reducing control and data dependences, and using speculation—are the same techniques we saw exploited in Chapter 3 using dynamic techniques. The key difference is that the techniques in this appendix are applied at compile time by the compiler, rather than at runtime by the hardware. The advantages of compile time techniques are primarily two: They do not burden runtime execution with any inefficiency, and they can take into account a wider range of the program than a runtime approach might be able to incorporate. As an example of the latter, the next section shows how a compiler might determine that an entire loop can be executed in parallel; hardware techniques might or might not be able to find such parallelism. The major disadvantage of static approaches is that they can use only compile time information. Without runtime information, compile time techniques must often be conservative and assume the worst case.

H.2 Detecting and Enhancing Loop-Level Parallelism

Loop-level parallelism is normally analyzed at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by renaming techniques like those we explored in Chapter 3.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such a dependence is called a *loop-carried dependence*. Most of the examples we considered in Section 3.2 have no loop-carried dependences and, thus, are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

```
for (i=1000; i>0; i=i-1)
     x[i] = x[i] + s;
```

In this loop, there is a dependence between the two uses of `x[i]`, but this dependence is within a single iteration and is not loop carried. There is a dependence between successive uses of `i` in different iterations, which is loop carried, but this dependence involves an induction variable and can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 3.2, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level. Let's look at a more complex example.

Example Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
     A[i+1] = A[i] + C[i];   /* S1 */
     B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that `A`, `B`, and `C` are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

Answer There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration `i` computes `A[i+1]`, which is read in iteration `i+1`. The same is true of S2 for `B[i]` and `B[i+1]`.
2. S2 uses the value, `A[i+1]`, computed by S1 in the same iteration.

These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 is on an earlier iteration of S1, this dependence is loop carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence (S2 depending on S1) is within an iteration and is not loop carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 3.2, where unrolling was able to expose the parallelism.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

Example Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];   /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

Answer Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular: Neither statement depends on itself, and, although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.
2. On the first iteration of the loop, statement S1 depends on the value of `B[1]` computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

The dependence between the two statements is no longer loop carried, so iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact*, in the sense that it tells us that such a dependence *may* exist. Consider the following example:

```
for (i=1;i<=100;i=i+1) {
      A[i] = B[i] + C[i]
      D[i] = A[i] * E[i]
}
```

The second reference to `A` in this example need not be translated to a load instruction, since we know that the value is computed and stored by the previous statement; hence, the second reference to `A` can simply be a reference to the register into which `A` was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis only tells that one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the example above, a simple version of this analysis suffices, since the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence:*

```
for (i=2;i<=100;i=i+1) {
      Y[i] = Y[i-1] + Y[i];
}
```

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the above fragment. Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and some recurrences can be the source of a reasonable amount of parallelism. To see how the latter can be true, consider this loop:

```
for (i=6;i<=100;i=i+1) {
      Y[i] = Y[i-5] + Y[i];
}
```

On the iteration i, the loop references element $i - 5$. The loop is said to have a *dependence distance* of 5. Many loops with carried dependences have a dependence distance of 1. The larger the distance, the more potential parallelism can be obtained by unrolling the loop. For example, if we unroll the first loop, with a dependence distance of 1, successive statements are dependent on one another; there is still some parallelism among the individual instructions, but not much. If we unroll the loop that has a dependence distance of 5, there is a sequence of five statements that have no dependences, and thus much more ILP. Although many loops with loop-carried dependences have a dependence distance of 1, cases with larger distances do arise, and the longer distance may well provide enough parallelism to keep a processor busy.

Finding Dependences

Finding the dependences in a program is an important part of three tasks: (1) good scheduling of code, (2) determining which loops might contain parallelism, and (3) eliminating name dependences. The complexity of dependence analysis arises because of the presence of arrays and pointers in languages like C or C++, or pass-by-reference parameter passing in FORTRAN. Since scalar variable references explicitly refer to a name, they can usually be analyzed quite easily, with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine*. In simplest terms, a one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form `x[y[i]]`, are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where i is the for-loop index variable that runs from m to n. A dependence exists if two conditions hold:

1. There are two iteration indices, j and k, both within the limits of the for loop. That is, $m \leq j \leq n$, $m \leq k \leq n$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that *same* array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether a dependence exists at compile time. For example, the values of *a*, *b*, *c*, and *d* may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where *a, b, c*, and *d* are all constants. For these cases, it is possible to devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (GCD) test. It is based on the observation that if a loop-carried dependence exists, then GCD (c,a) must divide ($d - b$). (Recall that an integer, *x, divides* another integer, *y*, if we get an integer quotient when we do the division *y*/*x* and there is no remainder.)

Example Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=1; i<=100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
}
```

Answer Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then GCD(a,c) = 2, and $d - b = -3$. Since 2 does not divide −3, no dependence is possible.

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

In general, determining whether a dependence actually exists is NP complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

Example The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=1; i<=100; i=i+1) {
    Y[i] = X[i] / c; /* S1 */
    X[i] = X[i] + c; /* S2 */
    Z[i] = Y[i] + c; /* S3 */
    Y[i] = c - Y[i]; /* S4 */
}
```

Answer The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of `Y[i]`. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on `X[i]`.
3. There is an antidependence from S3 to S4 for `Y[i]`.
4. There is an output dependence from S1 to S4, based on `Y[i]`.

The following version of the loop eliminates these false (or pseudo) dependences:

```
for (i=1; i<=100; i=i+1 {
    /* Y renamed to T to remove output dependence */
    T[i] = X[i] / c;
    /* X renamed to X1 to remove antidependence */
    X1[i] = X[i] + c;
    /* Y renamed to T to remove antidependence */
    Z[i] = T[i] + c;
    Y[i] = c - T[i];
}
```

After the loop, the variable `X` has been renamed `X1`. In code that follows the loop, the compiler can simply replace the name `X` by `X1`. In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

Dependence analysis is a critical technology for exploiting parallelism. At the instruction level, it provides information needed to interchange memory references when scheduling, as well as to determine the benefits of unrolling a loop. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs to either vector computers or multiprocessors depends critically on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances—namely, among references within a single loop nest and using affine index functions. Thus, there is a wide variety of situations in which array-oriented dependence analysis *cannot* tell us what we might want to know, including the following:

- When objects are referenced via pointers rather than array indices (but see discussion below)
- When array indexing is indirect through another array, which happens with many representations of sparse arrays
- When a dependence may exist for some value of the inputs but does not exist in actuality when the code is run since the inputs never take on those values
- When an optimization depends on knowing more than just the possibility of a dependence but needs to know on *which* write of a variable does a read of that variable depend

To deal with the issue of analyzing programs with pointers, another type of analysis, often called *points-to* analysis, is required (see Wilson and Lam [1995]). The key question that we want answered from dependence analysis of pointers is whether two pointers can designate the same address. In the case of complex dynamic data structures, this problem is extremely difficult. For example, we may want to know whether two pointers can reference the *same* node in a list at a given point in a program, which in general is undecidable and in practice is extremely difficult to answer. We may, however, be able to answer a simpler question: Can two pointers designate nodes in the *same* list, even if they may be separate nodes? This more restricted analysis can still be quite useful in scheduling memory accesses performed through pointers.

The basic approach used in points-to analysis relies on information from three major sources:

1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if p always points to an object allocated in a given source line and q never points to that object, then p and q can never point to the same object.
3. Information derived from pointer assignments. For example, if p may be assigned the value of q, then p may point to anything q points to.

There are several cases where analyzing pointers has been successfully applied and is extremely useful:

- When pointers are used to pass the address of an object as a parameter, it is possible to use points-to analysis to determine the possible set of objects referenced by a pointer. One important use is to determine if two pointer parameters may designate the same object.
- When a pointer can point to one of several types, it is sometimes possible to determine the type of the data object that a pointer designates at different parts of the program.
- It is often possible to separate out pointers that may only point to a local object versus a global one.

There are two different types of limitations that affect our ability to do accurate dependence analysis for large programs. The first type of limitation arises from restrictions in the analysis algorithms. Often, we are limited by the lack of applicability of the analysis rather than a shortcoming in dependence analysis *per se*. For example, dependence analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion—such as by doing arithmetic on pointers.

The second limitation is the need to analyze behavior across procedure boundaries to get accurate information. For example, if a procedure accepts two parameters that are pointers, determining whether the values could be the same requires analyzing across procedure boundaries. This type of analysis, called *interprocedural analysis*, is much more difficult and complex than analysis within a single procedure. Unlike the case of analyzing array indices within a single loop nest, points-to analysis usually requires an interprocedural analysis. The reason for this is simple. Suppose we are analyzing a program segment with two pointers; if the analysis does not know anything about the two pointers at the start of the program segment, it must be conservative and assume the worst case. The worst case is that the two pointers *may* designate the same object, but they are not *guaranteed* to designate the same object. This worst case is likely to propagate through the analysis, producing useless information. In practice, getting fully accurate interprocedural information is usually too expensive for real programs. Instead, compilers usually use approximations in interprocedural analysis. The result is that the information may be too inaccurate to be useful.

Modern programming languages that use strong typing, such as Java, make the analysis of dependences easier. At the same time the extensive use of procedures to structure programs, as well as abstract data types, makes the analysis more difficult. Nonetheless, we expect that continued advances in analysis algorithms, combined with the increasing importance of pointer dependency analysis, will mean that there is continued progress on this important problem.

Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more instruction-level parallelism (ILP). The key technique is to eliminate or reduce a dependent computation by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required. These techniques can be applied both within a basic block and within loops, and we describe them differently.

Within a basic block, algebraic simplifications of expressions and an optimization called *copy propagation*, which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI     R1,R2,#4
DADDUI     R1,R1,#4
```

to

```
DADDUI     R1,R2,#8
```

assuming this is the only use of `R1`. In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses in Section 3.2 are examples of this type of optimization.

In these examples, computations are actually eliminated, but it is also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called *tree height reduction* because they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

```
ADD        R1,R2,R3
ADD        R4,R1,R6
ADD        R8,R4,R7
```

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as

```
ADD        R1,R2,R3
ADD        R4,R6,R7
ADD        R8,R1,R4
```

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, for either integer arithmetic, because of limited range, or floating-point arithmetic, because of both range and precision. Thus, using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

When loops are unrolled, this sort of algebraic optimization is important to reduce the impact of dependences arising from recurrences. *Recurrences* are expressions whose value on one iteration is given by a function that depends on the previous iterations. When a loop with a recurrence is unrolled, we may be able to algebraically optimize the unrolled loop, so that the recurrence need only be evaluated once per unrolled iteration. One common type of recurrence arises from an explicit program statement, such as:

```
sum = sum + x;
```

Assume we unroll a loop with this recurrence five times. If we let the value of `x` on these five iterations be given by `x1`, `x2`, `x3`, `x4`, and `x5`, then we can write the value of `sum` at the end of each unroll as:

```
sum = sum + x1 + x2 + x3 + x4 + x5;
```

If unoptimized, this expression requires five dependent operations, but it can be rewritten as:

```
sum = ((sum + x1) + (x2 + x3)) + (x4 + x5);
```

which can be evaluated in only three dependent operations.

Recurrences also arise from implicit calculations, such as those associated with array indexing. Each array index translates to an address that is computed based on the loop index variable. Again, with unrolling and algebraic optimization, the dependent computations can be minimized.

H.3 Scheduling and Structuring Code for Parallelism

We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code. There are two other important techniques that have been developed for this purpose: *software pipelining* and *trace scheduling*.

Software Pipelining: Symbolic Loop Unrolling

Software pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the unrolled loop, which appeared in the example in Section 2.2. The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

A software-pipelined loop interleaves instructions from different iterations without unrolling the loop, as illustrated in Figure H.1. This technique is the software counterpart to what Tomasulo's algorithm does in hardware. The software-pipelined loop for the earlier example would contain one load, one add, and one store, each from a different iteration. There is also some start-up code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion, for simplicity.

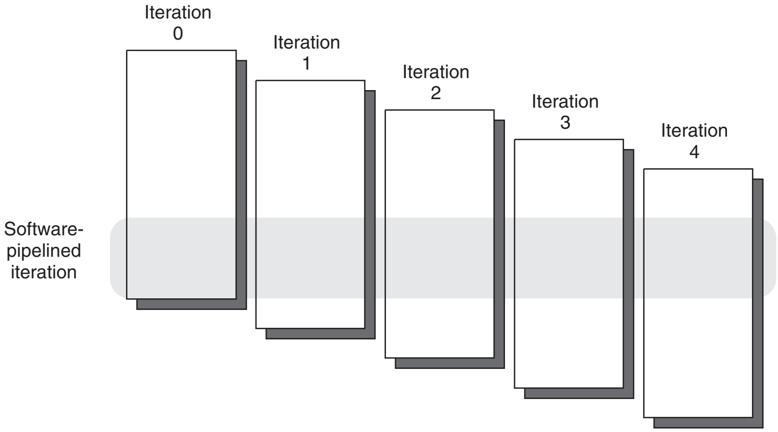


Figure H.1 A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

Example Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop:  L.D     F0,0(R1)
       ADD.D   F4,F0,F2
       S.D     F4,0(R1)
       DADDUI  R1,R1,#-8
       BNE     R1,R2,Loop
```

You may omit the start-up and clean-up code.

Answer Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the `DADDUI` and `BNE`) need not be replicated. Here's the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:

```
Iteration i:    L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
Iteration i+1:  L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
Iteration i+2:  L.D     F0,0(R1)
                ADD.D   F4,F0,F2
                S.D     F4,0(R1)
```

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

```
Loop:  S.D       F4,16(R1)     ;stores into M[i]
       ADD.D     F4,F0,F2      ;adds to M[i-1]
       L.D       F0,0(R1)      ;loads M[i-2]
       DADDUI    R1,R1,#-8
       BNE       R1,R2,Loop
```

This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that `DADDUI` is scheduled before the `ADD.D` and that the `L.D` instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations. Notice that the reuse of registers (e.g., F4, F0, and R1) requires the hardware to avoid the write after read (WAR) hazards that would occur in the loop. This hazard should not be a problem in this case, since no data-dependent stalls should occur.

By looking at the unrolled version we can see what the start-up code and finish-up code will need to be. For start-up, we will need to execute any instructions that correspond to iteration 1 and 2 that will not be executed. These instructions are the `L.D` for iterations 1 and 2 and the `ADD.D` for iteration 1. For the finish-up code, we need to execute any instructions that will not be executed in the final two iterations. These include the `ADD.D` for the last iteration and the `S.D` for the last two iterations.

Register management in software-pipelined loops can be tricky. The previous example is not too hard since the registers that are written on one loop iteration are read on the next. In other cases, we may need to increase the number of iterations between when we issue an instruction and when the result is used. This increase is required when there are a small number of instructions in the loop body and the latencies are large. In such cases, a combination of software pipelining and loop unrolling is needed.

Software pipelining can be thought of as *symbolic* loop unrolling. Indeed, some of the algorithms for software pipelining use loop-unrolling algorithms to figure out how to software-pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say, 4, we pay the overhead 100/4 = 25 times—every time the inner unrolled loop is initiated. Figure H.2 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance can come from doing both. In practice, compilation using software pipelining is quite difficult for several reasons: Many loops require significant transformation

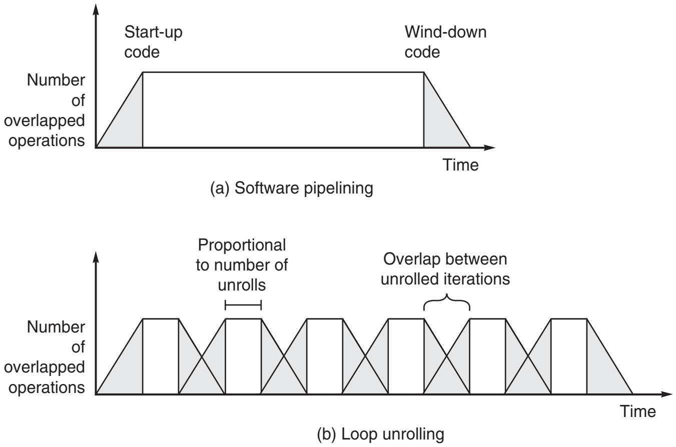

Start-up code
Wind-down code
Number of overlapped operations
Time
(a) Software pipelining
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
Time
(b) Loop unrolling

Figure H.2 The execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at the beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs *m/n* times if the loop has a total of *m* iterations and is unrolled *n* times. Each block represents an unroll of *n* iterations. Increasing the number of unrollings will reduce the start-up and clean-up overhead. The overhead of one iteration overlaps with the overhead of the next, thereby reducing the impact. The total area under the polygonal region in each case will be the same, since the total number of operations is just the execution rate multiplied by the time.

before they can be software pipelined, the trade-offs in terms of overhead versus efficiency of the software-pipelined loop are complex, and the issue of register management creates additional complexities. To help deal with the last two of these issues, the IA-64 added extensive hardware sport for software pipelining. Although this hardware can make it more efficient to apply software pipelining, it does not eliminate the need for complex compiler support, or the need to make difficult decisions about the best way to compile a loop.

Global Code Scheduling

In Section 3.2 we examined the use of loop unrolling and code scheduling to improve ILP. The techniques in Section 3.2 work well when the loop body is straight-line code, since the resulting unrolled loop looks like a single basic block. Similarly, software pipelining works well when the body is a single basic block, since it is easier to find the repeatable schedule. When the body of an unrolled loop contains internal control flow, however, scheduling the code is much more complex. In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of global code scheduling. In Section H.4 we examine hardware support for eliminating control flow within an inner loop, then we examine two compiler techniques that can be used when eliminating the control flow is not a viable approach.

Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved. Data dependences are overcome by unrolling and, in the case of memory operations, using dependence analysis to determine if two references refer to the same address. Finding the shortest possible sequence for a piece of code means finding the shortest sequence for the *critical path*, which is the longest sequence of dependent instructions.

Control dependences arising from loop branches are reduced by unrolling. Global code scheduling can reduce the effect of control dependences arising from conditional nonloop branches by moving code. Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths. Although global code motion cannot guarantee faster code, if the frequency information is accurate, the compiler can determine whether such code movement is likely to lead to faster code.

Global code motion is important since many inner loops contain conditional statements. Figure H.3 shows a typical code fragment, which may be thought of as an iteration of an unrolled loop, and highlights the more common control flow.

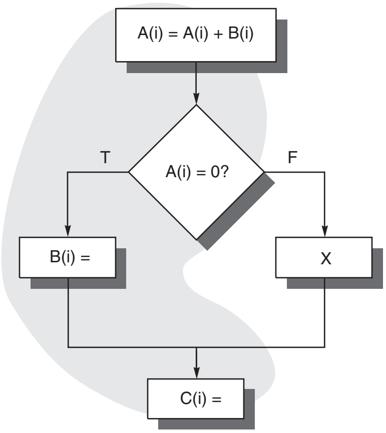


Figure H.3 A code fragment and the common path shaded with gray. Moving the assignments to B or C requires a more complex analysis than for straight-line code. In this section we focus on scheduling this code segment efficiently without hardware assistance. Predication or conditional instructions, which we discuss in the next section, provide another way to schedule this code.

Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative: It will speed the computation up only when the path containing the code would be taken.

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions. In Section H.5, we will see how hardware support allows for more opportunities for speculative code motion and removes control dependences. Although such enhanced support for speculation can make it possible to explore more opportunities, the difficulty of choosing how to best compile the code remains complex.

How can the compiler ensure that the assignments to B and C can be moved without affecting the data flow? To see what's involved, let's look at a typical code generation sequence for the flowchart in Figure H.3. Assuming that the addresses for A, B, C are in R1, R2, and R3, respectively, here is such a sequence:

```
            LD      R4,0(R1)        ;load A
            LD      R5,0(R2)        ;load B
            DADDU   R4,R4,R5        ;Add to A
            SD      R4,0(R1)        ;Store A
            ...
            BNEZ    R4,elsepart     ;Test A
            ...                     ;then part
            SD      ...,0(R2)       ;Stores to B
            ...
            J       join            ;jump over else
elsepart:   ...                     ;else part
            X                       ;code for X
            ...
join:       ...                     ;after if
            SD      ...,0(R3)       ;store C[i]
```

Let's first consider the problem of moving the assignment to B to before the `BNEZ` instruction. Call the last instruction to assign to B before the if statement *i*. If B is referenced before it is assigned either in code segment X or after the if statement, call the referencing instruction *j*. If there is such an instruction *j*, then moving the assignment to B will change the data flow of the program. In particular, moving the assignment to B will cause *j* to become data dependent on the moved version of the assignment to B rather than on *i*, on which *j* originally depended. You could imagine more clever schemes to allow B to be moved even when the value is used: For example, in the first case, we could make a shadow copy of B before the if statement and use that shadow copy in X. Such schemes are usually avoided, both because they are complex to implement and because they will slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions to execute.

Moving the assignment to C up to before the first branch requires two steps. First, the assignment is moved over the join point of the else part into the portion corresponding to the then part. This movement makes the instructions for C control dependent on the branch and means that they will not execute if the else path, which is the infrequent path, is chosen. Hence, instructions that were data dependent on the assignment to C, and which execute after this code fragment, will be affected. To ensure the correct value is computed for such instructions, a copy is made of the instructions that compute and assign to C on the else path. Second, we can move C from the then part of the branch across the branch condition, if it does not affect any data flow into the branch condition. If C is moved to before the if test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

We can see from this example that global code scheduling is subject to many constraints. This observation is what led designers to provide hardware support to make such code motion easier, and Sections H.4 and H.5 explores such support in detail.

Global code scheduling also requires complex trade-offs to make code motion decisions. For example, assuming that the assignment to B can be moved before the conditional branch (possibly with some compensation code on the alternative branch), will this movement make the code run faster? The answer is, possibly! Similarly, moving the copies of C into the if and else branches makes the code initially bigger! Only if the compiler can successfully move the computation across the if test will there be a likely benefit.

Consider the factors that the compiler would have to consider in moving the computation and assignment of B:

- What are the relative execution frequencies of the then case and the else case in the branch? If the then case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible, to consider moving the code.
- What is the cost of executing the computation and assignment to B above the branch? It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B "free" at least to first order.
- How will the movement of B change the execution time for the then case? If B is at the start of the critical path for the then case, moving it may be highly beneficial.
- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then case?
- What is the cost of the compensation code that may be necessary for the else case? How effectively can this code be scheduled, and what is its impact on execution time?

As we can see from this *partial* list, global code scheduling is an extremely complex problem. The trade-offs depend on many factors, and individual decisions to globally schedule instructions are highly interdependent. Even choosing which instructions to start considering as candidates for global code motion is complex!

To try to simplify this process, several different methods for global code scheduling have been developed. The two methods we briefly explore here rely on a simple principle: Focus the attention of the compiler on a straight-line code segment representing what is estimated to be the most frequently executed code path. Unrolling is used to generate the straight-line code, but, of course, the complexity arises in how conditional branches are handled. In both cases, they are effectively straightened by choosing and scheduling the most frequent path.

Trace Scheduling: Focusing on the Critical Path

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution (see Section H.4) is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths. Because it can generate *significant* overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input. Of course, this limits its effective applicability to certain classes of programs.

There are two steps to trace scheduling. The first step, called *trace selection,* tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a *trace*. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks. If, for example, the program fragment shown in Figure H.3 corresponds to an inner loop with the highlighted path being much more frequent, and the loop were unwound four times, the primary trace would consist of four copies of the shaded portion of the program, as shown in Figure H.4.

Once a trace is selected, the second process, called *trace compaction*, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to be the most probable path.

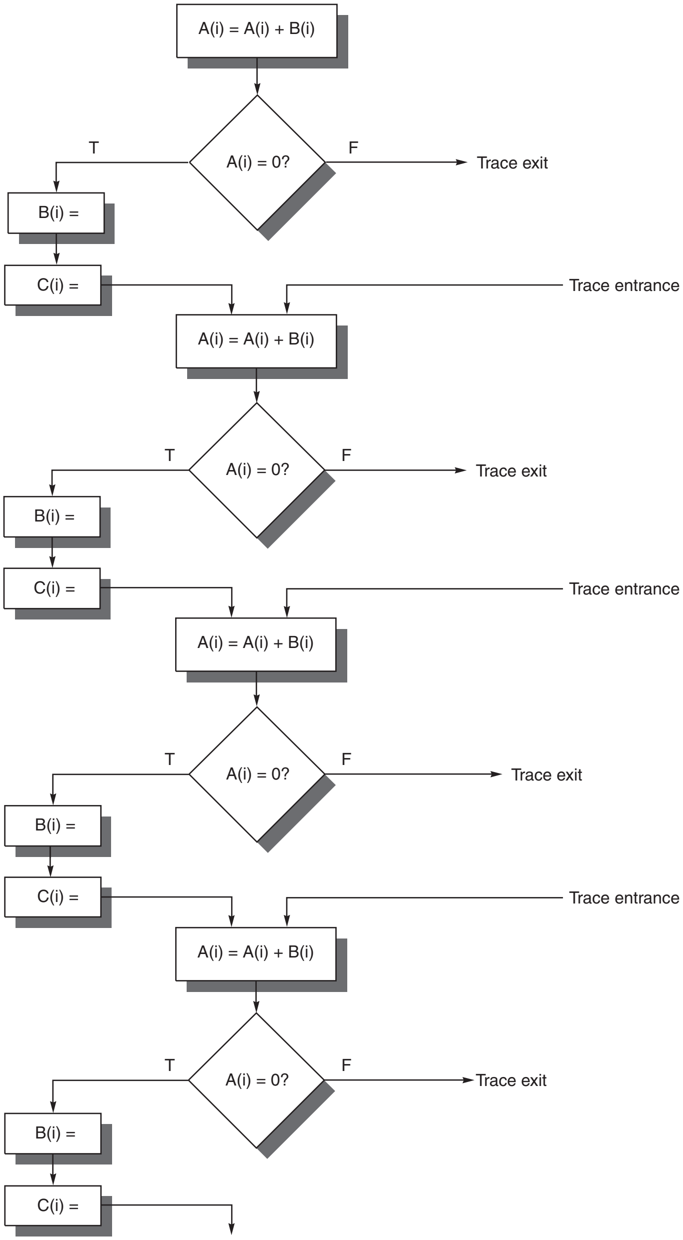

A(i) = A(i) + B(i)
A(i) = 0?
T
F
Trace exit
B(i) =
C(i) =
Trace entrance
A(i) = A(i) + B(i)
A(i) = 0?
T
F
Trace exit
B(i) =
C(i) =
Trace entrance
A(i) = A(i) + B(i)
A(i) = 0?
T
F
Trace exit
B(i) =
C(i) =
Trace entrance
A(i) = A(i) + B(i)
A(i) = 0?
T
F
Trace exit
B(i) =
C(i) =

Figure H.4 This trace is obtained by assuming that the program fragment in Figure H.3 is the inner loop and unwinding it four times, treating the shaded portion in Figure H.3 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point. The key assumption is that the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor: If an instruction can be moved and thereby make the main trace execute faster, it is moved.

Although trace scheduling has been successfully applied to scientific code with its intensive loops and accurate profile data, it remains unclear whether this approach is suitable for programs that are less simply characterized and less loop intensive. In such programs, the significant overheads of compensation code may make trace scheduling an unattractive approach, or, at best, its effective use will be extremely complex for the compiler.

Superblocks

One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications, requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code. *Superblocks* are formed by a process similar to that used for traces, but are a form of extended basic blocks, which are restricted to a single entry point but allow multiple exits.

Because superblocks have only a single entry point, compacting a superblock is easier than compacting a trace since only code motion across an exit need be considered. In our earlier example, we would form superblocks that contained only one entrance; hence, moving C would be easier. Furthermore, in loops that have a single loop exit based on a count (for example, a for loop with no loop exit other than the loop termination condition), the resulting superblocks have only one exit as well as one entrance. Such blocks can then be scheduled more easily.

How can a superblock with only one entrance be constructed? The answer is to use *tail duplication* to create a separate block that corresponds to the portion of the trace after the entry. In our previous example, each unrolling of the loop would create an exit from the superblock to a residual loop that handles the remaining iterations. Figure H.5 shows the superblock structure if the code fragment from Figure H.3 is treated as the body of an inner loop and unrolled four times. The residual loop handles any iterations that occur if the superblock is exited, which, in turn, occurs when the unpredicted path is selected. If the expected frequency of the residual loop were still high, a superblock could be created for that loop as well.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach but may enlarge code size more than a trace-based approach. Like trace scheduling, superblock scheduling may be most appropriate when other techniques (e.g., if conversion) fail. Even in such cases, assessing the cost of code duplication may limit the usefulness of the approach and will certainly complicate the compilation process.

Loop unrolling, software pipelining, trace scheduling, and superblock scheduling all aim at trying to increase the amount of ILP that can be exploited by a processor issuing more than one instruction on every clock cycle. The effectiveness of

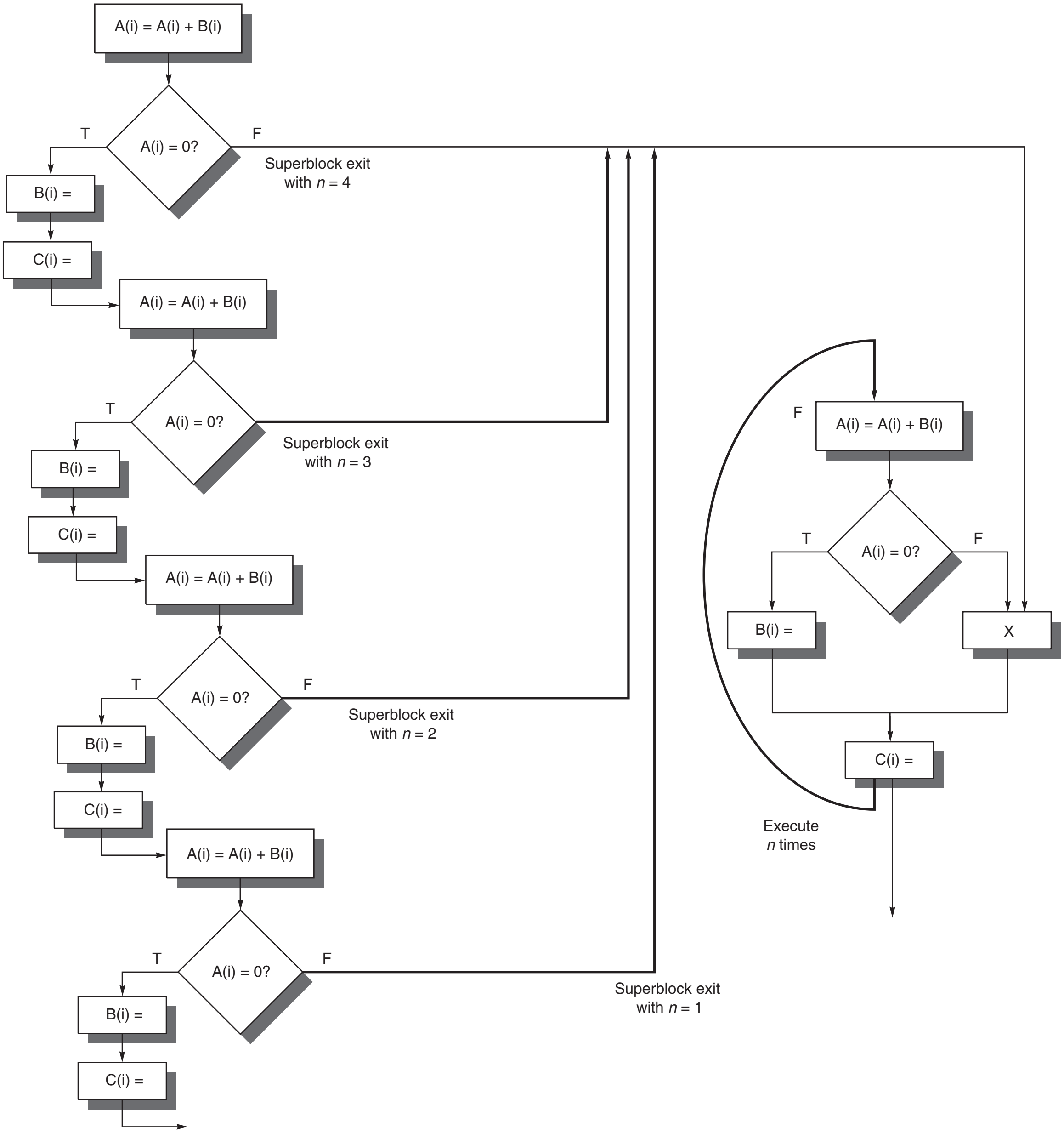

A(i) = A(i) + B(i)
T
A(i) = 0?
F
Superblock exit
with n = 4
B(i) =
C(i) =
A(i) = A(i) + B(i)
T
A(i) = 0?
Superblock exit
with n = 3
B(i) =
C(i) =
A(i) = A(i) + B(i)
T
A(i) = 0?
F
Superblock exit
with n = 2
B(i) =
C(i) =
A(i) = A(i) + B(i)
T
A(i) = 0?
F
Superblock exit
with n = 1
B(i) =
C(i) =
F
A(i) = A(i) + B(i)
T
A(i) = 0?
F
B(i) =
X
C(i) =
Execute
n times

Figure H.5 **This superblock results from unrolling the code in Figure H.3 four times and creating a superblock.**

each of these techniques and their suitability for various architectural approaches are among the hottest topics being actively pursued by researchers and designers of high-speed processors.

H.4 Hardware Support for Exposing Parallelism: Predicated Instructions

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. To overcome these problems, an architect can extend the instruction set to include *conditional* or *predicated instructions*. Such instructions can be used to eliminate branches, converting a control dependence into a data dependence and potentially improving performance. Such approaches are useful with either the hardware-intensive schemes in Chapter 3 or the software-intensive approaches discussed in this appendix, since in both cases predication can be used to eliminate branches.

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction were a no-op. Many newer architectures include some form of conditional instructions. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

Example Consider the following code:

```
if (A==0) {S=T;}
```

Assuming that registers `R1`, `R2`, and `R3` hold the values of `A`, `S`, and `T`, respectively, show the code for this statement with the branch and with the conditional move.

Answer The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

```
          BNEZ     R1,L
          ADDU     R2,R3,R0
L:
```

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
          CMOVZ    R2,R3,R1
```

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called *if conversion*.) For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.

One obvious use for conditional move is to implement the absolute value function: `A = abs (B)`, which is implemented as `if (B<0) {A=-B;} else {A=B;}`. This if statement can be implemented as a pair of conditional moves, or as one unconditional move `(A=B)` and one conditional move `(A=-B)`.

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement. Handling multiple branches per clock is complex, since one branch must be control dependent on the other. The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence has so far caused most designers to avoid processors that execute multiple branches per clock. Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.

Conditional moves are the simplest form of conditional or predicated instructions and, although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be inefficient, since many conditional moves may need to be introduced.

To remedy the inefficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then case executes only if the value of the condition is true and the code in the else case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly.

Example Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

First instruction slot		Second instruction slot
`LW`	`R1,40` `(R2)`	`ADD R3,R4,R5`
		`ADD R6,R3,R7`
`BEQZ`	`R10,L`	
`LW`	`R8,0` `(R10)`	
`LW`	`R9,0(R8)`	

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second `LW` after the branch depends on the prior load. Show how the code can be improved using a predicated form of `LW`.

Answer Call the predicated version load word `LWC` and assume the load occurs unless the third operand is 0. The `LW` immediately following the branch can be converted to an `LWC` and moved up to the second issue slot:

First instruction slot		Second instruction slot
`LW`	`R1,40(R2)`	`ADD R3,R4,R5`
`LWC`	`R8,0(R10),R10`	`ADD R6,R3,R7`
`BEQZ`	`R10,L`	
`LW`	`R9,0(R8)`	

This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.

If the sequence following the branch were short, the entire block of code might be converted to predicated execution and the branch eliminated.

When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false. The property of not causing exceptions is quite critical, as the previous example shows: If register `R10` contains zero, the instruction `LW R8,0(R10)` executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e., `R10` is not zero), the `LW` may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.

The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage. If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data-dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls. Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance. A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions, further complicating the implementation.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution time. Therefore, moving an instruction across a branch and making it conditional will slow the program down whenever the moved instruction would not have been normally executed. Likewise, predicating a control-dependent portion of code and eliminating a branch may slow down the processor if that code would not have been executed. An important exception to these situations occurs when the cycles used by the moved instruction when it is not performed would have been idle anyway (as in the earlier superscalar example). Moving an instruction across a branch or converting a code segment to predicated execution is essentially speculating on the outcome of the branch. Conditional instructions make this easier but do not eliminate the execution time taken by an incorrect guess. In simple cases, where we trade a conditional move for a branch and a move, using conditional moves or predication is almost always better. When longer code sequences are made conditional, the benefits are more limited.
- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be

separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.

- The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate. If such capabilities are not present, the overhead of if conversion will be larger, reducing its advantage.
- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, PowerPC, SPARC, and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions, as we will see in Section H.6.

H.5 Hardware Support for Compiler Speculation

As we saw in Chapter 3, many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if conversion. In many cases, we would like to move speculated instructions not only before the branch but also before the condition evaluation, and predication cannot achieve this.

To speculate ambitiously requires three capabilities:

1. The ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow
2. The ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur
3. The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts

The first of these is a compiler capability, while the last two require hardware support, which we explore next.

Hardware Support for Preserving Exception Behavior

To speculate ambitiously, we must be able to move any type of instruction and still preserve its exception behavior. The key to being able to do this is to observe that the results of a speculated sequence that is mispredicted will not be used in the final computation, and such a speculated instruction should not cause an exception.

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1. The hardware and operating system cooperatively ignore exceptions for speculative instructions. As we will see later, this approach preserves exception behavior for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a "fast mode" in several processors.
2. Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.
3. A set of status bits, called *poison bits*, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.
4. A mechanism is provided to indicate that an instruction is speculative, and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions. If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination. If the instruction generating the terminating exception was not speculative, then the program is in error. Note that instead of terminating the program, the program is allowed to continue, although it will almost certainly generate incorrect results. If the instruction generating the terminating exception is speculative, then the program may be correct and the speculative result will simply be unused; thus, returning an undefined value for the instruction cannot be harmful. This scheme can never cause a correct program to fail, no matter how much speculation is done. An incorrect program, which formerly might have received a terminating exception, will get an incorrect result. This is acceptable for some programs, assuming the compiler can also generate a normal version of the program, which does not speculate and can receive a terminating exception.

Example Consider that the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where `A` is at `0(R3)` and `B` is at `0(R2)`:

```
        LD      R1,0(R3)     ;load A
        BNEZ    R1,L1        ;test A
        LD      R1,0(R2)     ;then clause
        J       L2           ;skip else
L1:     DADDI   R1,R1,#4     ;else clause
L2:     SD      R1,0(R3)     ;store A
```

Assume that the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume `R14` is unused and available.

Answer Here is the new code:

```
        LD      R1,0(R3)     ;load A
        LD      R14,0(R2)    ;speculative load B
        BEQZ    R1,L3        ;other branch of the if
        DADDI   R14,R1,#4    ;the else clause
L3:     SD      R14,0(R3)    ;nonspeculative store
```

The then clause is completely speculated. We introduce a temporary register to avoid destroying `R1` when `B` is loaded; if the load is speculative, `R14` will be useless. After the entire code segment is executed, `A` will be in `R14`. The else clause could have also been compiled speculatively with a conditional move, but if the branch is highly predictable and low cost, this might slow the code down, since two extra instructions would always be executed as opposed to one branch.

In such a scheme, it is not necessary to know that an instruction is speculative. Indeed, it is helpful only when a program is in error and receives a terminating exception on a normal instruction; in such cases, if the instruction were not marked as speculative, the program could be terminated.

In this method for handling speculation, as in the next one, renaming will often be needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed and stores cannot be speculative. The small number of registers and the cost of spilling will act as one constraint on the amount of speculation. Of course, the major constraint remains the cost of executing speculative instructions when the compiler's branch prediction is incorrect.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions and instructions to check for such exceptions. This combination preserves the exception behavior exactly.

Example Show how the previous example can be coded using a speculative load (`sLD`) and a speculation check instruction (`SPECCK`) to completely preserve exception behavior. Assume `R14` is unused and available.

Answer Here is the code that achieves this:

```
        LD       R1,0(R3)     ;load A
        sLD      R14,0(R2)    ;speculative, no termination
        BNEZ     R1,L1        ;test A
        SPECCK   0(R2)        ;perform speculation check
        J        L2           ;skip else
L1:     DADDI    R14,R1,#4    ;else clause
L2:     SD       R14,0(R3)    ;store A
```

Notice that the speculation check requires that we maintain a basic block for the then case. If we had speculated only a portion of the then case, then a basic block representing the then case would exist in any event. More importantly, notice that checking for a possible exception requires extra code.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion. The scheme is simple: A poison bit is added to every register, and another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with a poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a register source with its poison bit turned on, the instruction causes a fault. In this way, any program that would have generated an exception still generates one, albeit at the first instance where a result is used by an instruction that is not speculative. Since poison bits exist only on register values and not memory values, stores are never speculative and thus trap if either operand is "poison."

Example Consider the code fragment from page H-29 and show how it would be compiled with speculative instructions and poison bits. Show where an exception for the speculative memory reference would be recognized. Assume `R14` is unused and available.

Answer Here is the code (an `s` preceding the opcode indicates a speculative instruction):

```
        LD       R1,0(R3)      ;load A
        sLD      R14,0(R2)     ;speculative load B
        BEQZ     R1,L3         ;
        DADDI    R14,R1,#4     ;
L3:     SD       R14,0(R3)     ;exception for speculative LW
```

If the speculative `sLD` generates a terminating exception, the poison bit of `R14` will be turned on. When the nonspeculative `SW` instruction occurs, it will raise an exception if the poison bit for `R14` is on.

One complication that must be overcome is how the OS saves the user registers on a context switch if the poison bit is set. A special instruction is needed to save and reset the state of the poison bits to avoid this problem.

The fourth and final approach listed earlier relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed. This last piece of information basically tells the hardware the location of the code block where the speculated instruction originally was. In practice, most of the benefit of speculation is gained by allowing movement across a single branch; thus, only 1 bit saying whether the speculated instruction came from the taken or not taken path is required. Alternatively, the original location of the speculative instruction is marked by a *sentinel*, which tells the hardware that the earlier speculative instruction is no longer speculative and values may be committed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. (Notice, though, that no actual speculative branch prediction or dynamic scheduling occurs.) The reorder buffer tracks when instructions are ready to commit and delays the "write-back" portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches that have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached. At that point, we know whether the speculated instruction should have been executed or not. If it should have been executed and it generated a terminating exception, then we know that the program should be terminated. If the instruction should not have been executed, then the exception can be ignored. Notice that the compiler, rather than the hardware, has the job of register renaming to ensure correct usage of the speculated result, as well as correct program execution.

Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. As we saw with the examples in Section 3.2, such transformations are critical to reducing the critical path length of a code segment. To allow the compiler to undertake such code motion when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian), and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched, then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction (which could supply the target register in addition to the memory address). If additional instructions that depended on the load were also speculated, then a fix-up sequence that reexecutes all the speculated instructions starting with the load is needed. In this case, the check instruction specifies the address where the fix-up code is located.

In this section, we have seen a variety of hardware assist mechanisms. Such mechanisms are key to achieving good support with the compiler-intensive approaches of Chapter 3 and this appendix. In addition, several of them can be easily integrated in the hardware-intensive approaches of Chapter 3 and provide additional benefits.

H.6 The Intel IA-64 Architecture and Itanium Processor

This section is an overview of the Intel IA-64 architecture, the most advanced VLIW-style processor, and its implementation in the Itanium processor.

The Intel IA-64 Instruction Set Architecture

The IA-64 is a RISC-style, register-register instruction set, but with many novel features designed to support compiler-based exploitation of ILP. Our focus here is on the unique aspects of the IA-64 ISA. Most of these aspects have been discussed already in this appendix, including predication, compiler-based parallelism detection, and support for memory reference speculation.

When they announced the IA-64 architecture, HP and Intel introduced the term EPIC (Explicitly Parallel Instruction Computer) to distinguish this new architectural approach from the earlier VLIW architectures and from other RISC architectures. Although VLIW and EPIC architectures share many features, the EPIC approach includes several concepts that extend the earlier VLIW approach. These extensions fall into two main areas:

1. EPIC has greater flexibility in indicating parallelism among instructions and in instruction formats. Rather than relying on a fixed instruction format where all operations in the instruction must be capable of being executed in parallel and where the format is completely rigid, EPIC uses explicit indicators of possible instruction dependence as well as a variety of instruction formats. This EPIC approach can express parallelism more flexibly than the more rigid VLIW method and can reduce the increases in code size caused by the typically inflexible VLIW instruction format.
2. EPIC has more extensive support for software speculation than the earlier VLIW schemes that had only minimal support.

In addition, the IA-64 architecture includes a variety of features to improve performance, such as register windows and a rotating floating-point register (FPR) stack.

The IA-64 Register Model

The components of the IA-64 register state are

- 128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide
- 128 82-bit floating-point registers, which provide two extra exponent bits over the standard 80-bit IEEE format
- 64 1-bit predicate registers
- 8 64-bit branch registers, which are used for indirect branches
- A variety of registers used for system control, memory mapping, performance counters, and communication with the OS

The integer registers are configured to help accelerate procedure calls using a register stack mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture. Registers 0 to 31 are always accessible and are addressed as 0 to 31. Registers 32 to 128 are used as a register stack, and each procedure is allocated a set of registers (from 0 to 96) for its use. The new register stack frame is created for a called procedure by renaming the registers in hardware; a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure. The frame consists of two parts: the local area and the output area. The local area is used for local storage, while the output area is used to pass values to any called procedure. The `alloc` instruction specifies the size of these areas. Only the integer registers have register stack support.

On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure. This update enables the parameters of the caller to be passed into the addressable registers of the callee. The callee executes an `alloc` instruction to allocate both the number of required local registers, which include the output registers of the caller, and the number of output registers needed for parameter passing to a called procedure. Special load and store instructions are available for saving and restoring the register stack, and special hardware (called the *register stack engine*) handles overflow of the register stack.

In addition to the integer registers, there are three other sets of registers: the floating-point registers, the predicate registers, and the branch registers. The floating-point registers are used for floating-point data, and the branch registers are used to hold branch destination addresses for indirect branches. The predication registers hold predicates, which control the execution of predicated instructions; we describe the predication mechanism later in this section.

Both the integer and floating-point registers support register rotation for registers 32 to 128. Register rotation is designed to ease the task of allocating registers in software-pipelined loops, a problem that we discussed in Section H.3. In addition, when combined with the use of predication, it is possible to avoid the need for unrolling and for separate prologue and epilogue code for a software-pipelined loop. This capability reduces the code expansion incurred to use software pipelining and makes the technique usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.

Instruction Format and Support for Explicit Parallelism

The IA-64 architecture is designed to achieve the major benefits of a VLIW approach—implicit parallelism among operations in an instruction and fixed formatting of the operation fields—while maintaining greater flexibility than a VLIW normally allows. This combination is achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots, but adding flexibility in the formatting of instructions and allowing the compiler to indicate when an instruction cannot be executed in parallel with its successors.

The IA-64 architecture uses two different concepts to achieve the benefits of implicit parallelism and ease of instruction decode. Implicit parallelism is achieved by placing instructions into *instruction groups*, while the fixed formatting of multiple instructions is achieved through the introduction of a concept called a *bundle*, which contains three instructions. Let's start by defining an instruction group.

An instruction group is a sequence of consecutive instructions with no register data dependences among them (there are a few minor exceptions). All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate the

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	Add, subtract, and, or, compare
	I	Non-ALU	integer Integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	Add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

Figure H.6 The five execution unit slots in the IA-64 architecture and what instructions types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either an I-unit or M-unit slot. L + X slots are special, as they occupy two instruction slots; L + X instructions are used to encode 64-bit immediates and a few special instructions. L + X instructions are executed either by the I-unit or the B-unit.

boundary between one instruction group and another. This boundary is indicated by placing a *stop* between two instructions that belong to different groups. To understand how stops are indicated, we must first explain how instructions are placed into bundles.

IA-64 instructions are encoded in bundles, which are 128 bits wide. Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. (Actually, the 41-bit quantities are not truly instructions, since they can only be interpreted in conjunction with the template field. The name *syllable* is sometimes used for these operations. For simplicity, we will continue to use the term "instruction.") To simplify the decoding and instruction issue process, the template field of a bundle specifies what types of execution units each instruction in the bundle requires. Figure H.6 shows the five different execution unit types and describes what instruction classes they may hold, together with some examples.

The 5-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. Figure H.7 shows the possible formats that the template field encodes and the position of any stops it specifies. The bundle formats can specify only a subset of all possible combinations of instruction types and stops. To see how the bundle works, let's consider an example.

Example Unroll the array increment example, `x[i] = x[i] + s`, seven times and place the instructions into bundles, first ignoring pipeline latencies (to minimize the number of bundles) and then scheduling the code to minimize stalls. In scheduling the code assume one bundle executes per clock and that any stalls cause the entire bundle to

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

Figure H.7 The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines and may appear within and/or at the end of the bundle. For example, template 9 specifies that the instruction slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instruction slots but also includes a stop after the first slot. The L + X format is used when slot 1 is L and slot 2 is X.

be stalled. Use the pipeline latencies from Figure 3.2. Use MIPS instruction mnemonics for simplicity.

Answer The two different versions are shown in Figure H.8. Although the latencies are different from those in Itanium, the most common bundle, MMF, must be issued by itself in Itanium, just as our example assumes.

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/ cycle)
9: M M I	`L.D F0,0(R1)`	`L.D F6,-8(R1)`		1
14: M M F	`L.D F10,-16(R1)`	`L.D F14,-24(R1)`	`ADD.D F4,F0,F2`	3
15: M M F	`L.D F18,-32(R1)`	`L.D F22,-40(R1)`	`ADD.D F8,F6,F2`	4
15: M M F	`L.D F26,-48(R1)`	`S.D F4,0(R1)`	`ADD.D F12,F10,F2`	6
15: M M F	`S.D F8,-8(R1)`	`S.D F12,-16(R1)`	`ADD.D F16,F14,F2`	9
15: M M F	`S.D F16,-24(R1)`		`ADD.D F20,F18,F2`	12
15: M M F	`S.D F20,-32(R1)`		`ADD.D F24,F22,F2`	15
15: M M F	`S.D F24,-40(R1)`		`ADD.D F28,F26,F2`	18
16: M I B	`S.D F28,-48(R1)`	`DADDUI R1,R1,#-56`	`BNE R1,R2,Loop`	21

(a) The code scheduled to minimize the number of bundles

Bundle template	Slot 0	Slot 1	Slot 2	Execute cycle (1 bundle/ cycle)
8: M M I	`L.D F0,0(R1)`	`L.D F6,-8(R1)`		1
9: M M I	`L.D F10,-16(R1)`	`L.D F14,-24(R1)`		2
14: M M F	`L.D F18,-32(R1)`	`L.D F22,-40(R1)`	`ADD.D F4,F0,F2`	3
14: M M F	`L.D F26,-48(R1)`		`ADD.D F8,F6,F2`	4
15: M M F			`ADD.D F12,F10,F2`	5
14: M M F		`S.D F4,0(R1)`	`ADD.D F16,F14,F2`	6
14: M M F		`S.D F8,-8(R1)`	`ADD.D F20,F18,F2`	7
15: M M F		`S.D F12,-16(R1)`	`ADD.D F24,F22,F2`	8
14: M M F		`S.D F16,-24(R1)`	`ADD.D F28,F26,F2`	9
9: M M I	`S.D F20,-32(R1)`	`S.D F24,-40(R1)`		11
16: M I B	`S.D F28,-48(R1)`	`DADDUI R1,R1,#-56`	`BNE R1,R2,Loop`	12

(b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle

Figure H.8 The IA-64 instructions, including bundle bits and stops, for the unrolled version of x[i] = x[i] + s, when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instruction slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures. Note that the branch in the last bundle in both sequences depends on the `DADD` in the same bundle. In the IA-64 instruction set, this sequence would be coded as a setting of a predication register and a branch that would be predicated on that register. Normally, such dependent operations could not occur in the same bundle, but this case is one of the exceptions mentioned earlier.

Instruction Set Basics

Before turning to the special support for speculation, we briefly discuss the major instruction encodings and survey the instructions in each of the five primary instruction classes (A, I, M, F, and B). Each IA-64 instruction is 41 bits in length. The high-order 4 bits, together with the bundle bits that specify the execution unit slot, are used as the major opcode. (That is, the 4-bit opcode field is reused across the execution field slots, and it is appropriate to think of the opcode as being 4 bits plus the M, F, I, B, L + X designation.) The low-order 6 bits of every instruction are used for specifying the predicate register that guards the instruction (see the next subsection).

Figure H.9 summarizes most of the major instruction formats, other than the multimedia instructions, and gives examples of the instructions encoded for each format.

Predication and Speculation Support

The IA-64 architecture provides comprehensive support for predication: Nearly every instruction in the IA-64 architecture can be predicated. An instruction is predicated by specifying a predicate register, whose identity is placed in the lower 6 bits of each instruction field. Because nearly all instructions can be predicated, both if conversion and code motion have lower overhead than they would with only limited support for conditional instructions. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

Predicate registers are set using compare or test instructions. A compare instruction specifies one of ten different comparison tests and two predicate registers as destinations. The two predicate registers are written either with the result of the comparison (0 or 1) and the complement, or with some logical function that combines the two tests (such as `and`) `and` the complement. This capability allows multiple comparisons to be done in one instruction.

Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exception for speculated instructions, and memory reference speculation, which supports speculation of load instructions.

Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits. For the general-purpose registers (GPRs), these bits are called NaTs (Not a Thing), and this extra bit makes the GPRs effectively 65 bits wide. For the FP registers this capability is obtained using a special value, NaTVal (Not a Thing Value); this value is encoded using a significand of 0 and an exponent outside of the IEEE range. Only speculative load instructions generate such values, but all instructions that do not affect memory will cause a NaT or NaTVal to be propagated to the result register. (There are both speculative and non-speculative loads; the latter can only raise immediate exceptions and cannot defer them.) Floating-point exceptions are not handled through this mechanism but instead use floating-point status registers to record exceptions.

Instruction type	Number of formats	Representative instructions	Extra opcode bits	GPRs/ FPRs	Immediate bits	Other/comment
A	8	Add, subtract, and, or	9	3	0	
		Shift left and add	7	3	0	2-bit shift count
		ALU immediates	9	2	8	
		Add immediate	3	2	14	
		Add immediate	0	2	22	
		Compare	4	2	0	2 predicate register destinations
		Compare immediate	3	1	8	2 predicate register destinations
I	29	Shift R/L variable	9	3	0	Many multimedia instructions use this format.
		Test bit	6	3	6-bit field specifier	2 predicate register destinations
		Move to BR	6	1	9-bit branch predict	Branch register specifier
M	46	Integer/FP load and store, line prefetch	10	2	0	Speculative/ nonspeculative
		Integer/FP load and store, and line prefetch and post-increment by immediate	9	2	8	Speculative/ nonspeculative
		Integer/FP load prefetch and register postincrement	10	3		Speculative/ nonspeculative
		Integer/FP speculation check	3	1	21 in two fields	
B	9	PC-relative branch, counted branch	7	0	21	
		PC-relative call	4	0	21	1 branch register
F	15	FP arithmetic	2	4		
		FP compare	2	2		2 6-bit predicate regs
L + X	4	Move immediate long	2	1	64	

Figure H.9 A summary of some of the instruction formats of the IA-64 ISA. The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: A different use of a field, even of the same size, is considered a different format. The number of formats that actually have *different field sizes* is one-third to one-half as large. Some instructions have unused bits that are reserved; we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. Only one of the many formats for the multimedia instructions is shown in this table.

A deferred exception can be resolved in two different ways. First, if a nonspeculative instruction, such as a store, receives a NaT or NaTVal as a source operand, it generates an immediate and unrecoverable exception. Alternatively, a `chk.s` instruction can be used to detect the presence of NaT or NaTVal and branch to a routine designed by the compiler to recover from the speculative operation. Such a recovery approach makes more sense for memory reference speculation.

The inability to store the contents of instructions with a NaT or NaTVal set would make it impossible for the OS to save the state of the processor. Thus, IA-64 includes special instructions to save and restore registers that do not cause an exception for a NaT or NaTVal and also save and restore the NaT bits.

Memory reference support in the IA-64 uses a concept called *advanced loads*. An advanced load is a load that has been speculatively moved above store instructions on which it is potentially dependent. To speculatively perform a load, the `ld.a` (for advanced load) instruction is used. Executing this instruction creates an entry in a special table, called the *ALAT*. The ALAT stores both the register destination of the load and the address of the accessed memory location. When a store is executed, an associative lookup against the active ALAT entries is performed. If there is an ALAT entry with the same memory address as the store, the ALAT entry is marked as invalid.

Before any nonspeculative instruction (i.e., a store) uses the value generated by an advanced load or a value derived from the result of an advanced load, an explicit check is required. The check specifies the destination register of the advanced load. If the ALAT for that register is still valid, the speculation was legal and the only effect of the check is to clear the ALAT entry. If the check fails, the action taken depends on which of two different types of checks was employed. The first type of check is an instruction `ld.c`, which simply causes the data to be reloaded from memory at that point. An `ld.c` instruction is used when *only* the load is advanced. The alternative form of a check, `chk.a`, specifies the address of a fix-up routine that is used to reexecute the load *and any other* speculated code that depended on the value of the load.

The Itanium 2 Processor

The Itanium 2 processor is the second implementation of the IA-64 architecture. The first version, Itanium 1, became available in 2001 with an 800 MHz clock. The Itanium 2, first delivered in 2003, had a maximum clock rate in 2005 of 1.6 GHz. The two processors are very similar, with some differences in the pipeline structure and greater differences in the memory hierarchies. The Itanium 2 is about four times faster than the Itanium 1. This performance improvement comes from a doubling of the clock rate, a more aggressive memory hierarchy, additional functional units that improve instruction throughput, more complete bypassing, a shorter pipeline that reduces some stalls, and a more mature compiler system. During roughly the same period that elapsed from the Itanium 1 to Itanium 2, the Pentium processors improved by slightly more than a factor of three. The greater improvement for the Itanium is reasonable given the novelty of the architecture and software system versus the more established IA-32 implementations.

The Itanium 2 can fetch and issue two bundles, or up to six instructions, per clock. The Itanium 2 uses a three-level memory hierarchy all on-chip. The first level uses split instruction and data caches, each 16 KB; floating-point data are not placed in the first-level cache. The second and third levels are unified caches of 256 KB and of 3 MB to 9 MB, respectively.

Functional Units and Instruction Issue

There are 11 functional units in the Itanium 2 processor: two I-units, four M-units (two for loads and two for stores), three B-units, and two F-units. All the functional units are pipelined. Figure H.10 gives the pipeline latencies for some typical instructions. In addition, when a result is bypassed from one unit to another, there is usually at least one additional cycle of delay.

Itanium 2 can issue up to six instructions per clock from two bundles. In the worst case, if a bundle is split when it is issued, the hardware could see as few as four instructions: one from the first bundle to be executed and three from the second bundle. Instructions are allocated to functional units based on the bundle bits, ignoring the presence of no-ops or predicated instructions with untrue predicates. In addition, when issue to a functional unit is blocked because the next instruction to be issued needs an already committed unit, the resulting bundle is split. A split bundle still occupies one of the two bundle slots, even if it has only one instruction remaining.

Instruction	**Latency**
Integer load	1
Floating-point load	5–9
Correctly predicted taken branch	0–3
Mispredicted branch	6
Integer ALU operations	0
FP arithmetic	4

Figure H.10 The latency of some typical instructions on Itanium 2. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is equal to the access time of the second-level cache. There are some minor restrictions for some of the functional units, but these primarily involve the execution of infrequent instructions.

The Itanium 2 processor uses an eight-stage pipeline divided into four major parts:

- *Front-end (stages IPG and Rotate)*—Prefetches up to 32 bytes per clock (two bundles) into a prefetch buffer, which can hold up to eight bundles (24 instructions). Branch prediction is done using a multilevel adaptive predictor like those described in Chapter 3.
- *Instruction delivery (stages EXP and REN)*—Distributes up to six instructions to the 11 functional units. Implements register renaming for both rotation and register stacking.
- *Operand delivery (REG)*—Accesses the register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences. The scoreboard is used to detect when individual instructions can proceed, so that a stall of one instruction (for example, due to an unpredictable event like a cache miss) in a bundle need not cause the entire bundle to stall. (As we saw in Figure H.8, stalling the entire bundle leads to poor performance unless the instructions are carefully scheduled.)
- *Execution (EXE, DET, and WRB)*—Executes instructions through ALUs and load-store units, detects exceptions and posts NaTs, retires instructions, and performs write-back.

Both the Itanium 1 and the Itanium 2 have many of the features more commonly associated with the dynamically scheduled pipelines described in Chapter 3: dynamic branch prediction, register renaming, scoreboarding, a pipeline with a number of stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection. Although these mechanisms are generally simpler than those in an advanced dynamically scheduled superscalar, the overall effect is that the Itanium processors, which rely much more on compiler technology, seem to be as complex as the dynamically scheduled processors we saw in Chapter 3!

One might ask why such features are included in a processor that relies primarily on compile time techniques for finding and exploiting parallelism. There are two main motivations. First, dynamic techniques are sometimes significantly better, and omitting them would hurt performance significantly. The inclusion of dynamic branch prediction is such a case.

Second, caches are absolutely necessary to achieve high performance, and with caches come cache misses, which are both unpredictable and which in current processors take a relatively long time. In the early VLIW processors, the entire processor would freeze when a cache miss occurred, retaining the lockstep parallelism initially specified by the compiler. Such an approach is totally unrealistic in a modern processor where cache misses can cost tens to hundreds of cycles. Allowing some instructions to continue while others are stalled, however, requires the introduction of some form of dynamic scheduling, in this case scoreboarding. In addition, if a stall is likely to be long, then antidependences are likely to prevent much

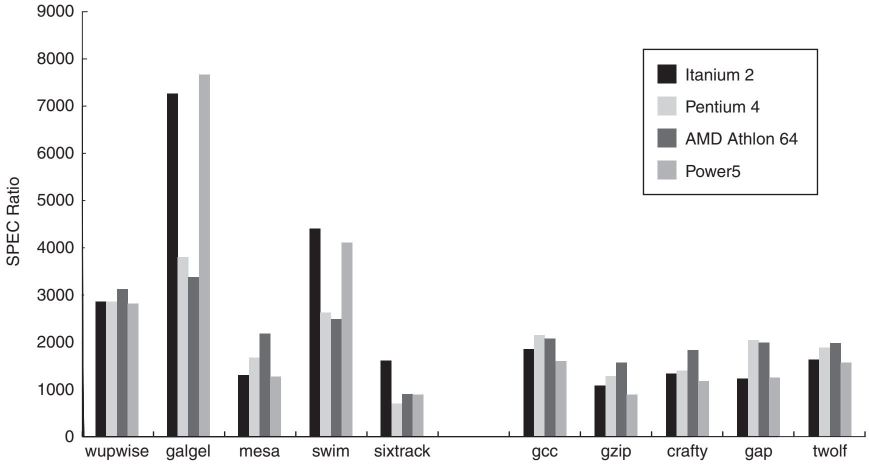


Figure H.11 The performance of four multiple-issue processors for five SPECfp and SPECint benchmarks. The clock rates of the four processors are Itanium 2 at 1.5 GHz, Pentium 4 Extreme Edition at 3.8 GHz, AMD Athlon 64 at 2.8 GHz, and the IBM Power5 at 1.9 GHz.

progress while waiting for the cache miss; hence, the Itanium implementations also introduce register renaming.

Itanium 2 Performance

Figure H.11 shows the performance of a 1.5 GHz Itanium 2 versus a Pentium 4, an AMD Athlon processor, and an IBM Power5 for five SPECint and five SPECfp benchmarks. Overall, the Itanium 2 is slightly slower than the Power5 for the full set of SPEC floating-point benchmarks and about 35% faster than the AMD Athlon or Pentium 4. On SPECint, the Itanium 2 is 15% faster than the Power5, while both the AMD Athlon and Pentium 4 are about 15% faster than the Itanium 2. The Itanium 2 and Power5 are much higher power and have larger die sizes. In fact, the Power5 contains two processors, only one of which is active during normal SPEC benchmarks, and still it has less than half the transistor count of the Itanium. If we were to reduce the die size, transistor count, and power of the Power5 by eliminating one of the processors, the Itanium would be by far the largest and highest-power processor.

H.7 Concluding Remarks

When the design of the IA-64 architecture began, it was a joint effort of Hewlett-Packard and Intel and many of the designers had benefited from experience with early VLIW processors as well of years of research building on the early concepts. The clear goal for the IA-64 architecture was to achieve levels of ILP as good or better than what had been achieved with hardware-based approaches, while also allowing a much simpler hardware implementation. With a simpler hardware implementation, designers hoped that much higher clock rates could be achieved. Indeed, when the IA-64 architecture and the first Itanium were announced, they were announced as the successor to the RISC approaches with clearly superior advantages.

Unfortunately, the practical reality has been quite different. The IA-64 and Itanium implementations appear to be at least as complicated as the dynamically based speculative processors, and neither approach has a significant and consistent performance advantage. The fact that the Itanium designs have also not been more power efficient has led to a situation where the Itanium design has been adopted by only a small number of customers primarily interested in FP performance.

Intel had planned for IA-64 to be its new 64-bit architecture as well. But the combination of its mediocre integer performance (especially in Itanium 1) and large die size, together with AMD's introduction of a 64-bit version of the IA-32 architecture, forced Intel to extend the address space of IA-32. The availability of a larger address space IA-32 processor with strong integer performance has further reduced the interest in IA-64 and Itanium. Most recently, Intel has introduced the name IPF to replace IA-64, since the former name made less sense once the older x86 architecture was extended to 64 bits.

Reference

Wilson, R.P., Lam, M.S., 1995. Efficient context-sensitive pointer analysis for C programs. In: Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation, June 18–21, 1995, La Jolla, Calif, pp. 1–12.

I.1 Introduction I-2
I.2 Interprocessor Communication: The Critical Performance Issue I-3
I.3 Characteristics of Scientific Applications I-6
I.4 Synchronization: Scaling Up I-12
I.5 Performance of Scientific Applications on Shared-Memory Multiprocessors I-21
I.6 Performance Measurement of Parallel Processors with Scientific Applications I-33
I.7 Implementing Cache Coherence I-34
I.8 The Custom Cluster Approach: Blue Gene/L I-41
I.9 Concluding Remarks I-44

I

Large-Scale Multiprocessors and Scientific Applications

Hennessy and Patterson should move MPPs to Chapter 11.

Jim Gray, Microsoft Research
when asked about the coverage of massively parallel processors (MPPs) for the third edition in 2000

Unfortunately for companies in the MPP business, the third edition had only ten chapters and the MPP business did not grow as anticipated when the first and second edition were written.

I.1 Introduction

The primary application of large-scale multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing where independent tasks are executed in parallel without much interaction. In true parallel computing, a set of tasks execute in a collaborative fashion on one application. The primary target of parallel computing is scientific and technical applications. In contrast, for loosely coupled commercial applications, such as Web servers and most transaction-processing applications, there is little communication among tasks. For such applications, loosely coupled clusters are generally adequate and most cost-effective, since intertask communication is rare.

Because true parallel computing involves cooperating tasks, the nature of communication between those tasks and how such communication is supported in the hardware is of vital importance in determining the performance of the application. The next section of this appendix examines such issues and the characteristics of different communication models.

In comparison to sequential programs, whose performance is largely dictated by the cache behavior and issues related to instruction-level parallelism, parallel programs have several additional characteristics that are important to performance, including the amount of parallelism, the size of parallel tasks, the frequency and nature of intertask communication, and the frequency and nature of synchronization. These aspects are affected both by the underlying nature of the application as well as by the programming style. Section I.3 reviews the important characteristics of several scientific applications to give a flavor of these issues.

As we saw in Chapter 5, synchronization can be quite important in achieving good performance. The larger number of parallel tasks that may need to synchronize makes contention involving synchronization a much more serious problem in large-scale multiprocessors. Section I.4 examines methods of scaling up the synchronization mechanisms of Chapter 5.

Section I.5 explores the detailed performance of shared-memory parallel applications executing on a moderate-scale shared-memory multiprocessor. As we will see, the behavior and performance characteristics are quite a bit more complicated than those in small-scale shared-memory multiprocessors. Section I.6 discusses the general issue of how to examine parallel performance for different sized multiprocessors. Section I.7 explores the implementation challenges of distributed shared-memory cache coherence, the key architectural approach used in moderate-scale multiprocessors. Sections I.7 and I.8 rely on a basic understanding of interconnection networks, and the reader should at least quickly review Appendix F before reading these sections.

Section I.8 explores the design of one of the newest and most exciting large-scale multiprocessors in recent times, Blue Gene. Blue Gene is a cluster-based multiprocessor, but it uses a custom, highly dense node designed specifically for this function, as opposed to the nodes of most earlier cluster multiprocessors that used a node architecture similar to those in a desktop or smaller-scale multiprocessor node. By using a custom node design, Blue Gene achieves a significant reduction in the cost, physical size, and power consumption of a node. Blue Gene/L, a 64 K-node version, was the world's fastest computer in 2006, as measured by the linear algebra benchmark, Linpack.

I.2 Interprocessor Communication: The Critical Performance Issue

In multiprocessors with larger processor counts, interprocessor communication becomes more expensive, since the distance between processors increases. Furthermore, in truly parallel applications where the threads of the application must communicate, there is usually more communication than in a loosely coupled set of distinct processes or independent transactions, which characterize many commercial server applications. These factors combine to make efficient interprocessor communication one of the most important determinants of parallel performance, especially for the scientific market.

Unfortunately, characterizing the communication needs of an application and the capabilities of an architecture is complex. This section examines the key hardware characteristics that determine communication performance, while the next section looks at application behavior and communication needs.

Three performance metrics are critical in any hardware communication mechanism:

1. *Communication bandwidth*—Ideally, the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The interconnection network determines the maximum communication capacity of the system. The bandwidth in or out of a single node, which is often as important as total system bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this *occupancy* is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.
2. *Communication latency*—Ideally, the latency is as low as possible. As Appendix F explains:

$$\begin{aligned}\text{Communication latency} &= \text{Sender overhead} + \text{Time of flight}\\ &\quad + \text{Transmission time} + \text{Receiver overhead}\end{aligned}$$

assuming no contention. Time of flight is fixed and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait.

Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the communication mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two metrics, it is an important characteristic that can be quantified by measuring the running time on multiprocessors with the same communication latency but different support for latency hiding. Although hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus, it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application, as we will see in the next section. The size of the data items being communicated is the most obvious characteristic, since it affects both latency and bandwidth directly, as well as affecting the efficacy of different latency-hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

Advantages of Different Communication Mechanisms

The two primary means of communicating data in a large-scale multiprocessor are message passing and shared memory. Each of these two primary communication mechanisms has its advantages. For shared-memory communication, the advantages include

- Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication. The OpenMP consortium (see *www.openmp.org* for description) has proposed a standardized programming interface for shared-memory multiprocessors. Although message passing also uses a standard, MPI or Message Passing Interface, this standard is not used either in shared-memory multiprocessors or in loosely coupled clusters in use in throughput-oriented environments.
- Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.
- The ability to develop applications using the familiar shared-memory model, focusing attention only on those accesses that are performance critical.
- Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the I/O system.
- The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data. This advantage also comes with a disadvantage, which we mention below.

The major advantages for message-passing communication include the following:

- The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.
- Communication is explicit, which means it is simpler to understand. In shared-memory models, it can be difficult to know when communication is occurring and when it is not, as well as how costly the communication is.
- Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improved structure in a multiprocessor program.
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
- It makes it easier to use sender-initiated communication, which may have some advantages in performance.
- If the communication is less frequent and more structured, it is easier to improve fault tolerance by using a transaction-like structure. Furthermore,

the less tight coupling of nodes and explicit communication make fault isolation simpler.

- The very largest multiprocessors use a cluster structure, which is inherently based on message passing. Using two different communication models may introduce more complexity than is warranted.

Of course, the desired communication model can be created in software on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. For these reasons, it has never been practical to use message passing to implement shared memory for a commercial system.

I.3 Characteristics of Scientific Applications

The primary use of scalable shared-memory multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing. The primary target of parallel computing is scientific and technical applications. Thus, understanding the design issues requires some insight into the behavior of such applications. This section provides such an introduction.

Characteristics of Scientific Applications

Our scientific/technical parallel workload consists of two applications and two computational kernels. The kernels are fast Fourier transformation (FFT) and an LU decomposition, which were chosen because they represent commonly used techniques in a wide variety of applications and have performance characteristics typical of many parallel scientific applications. In addition, the kernels have small code segments whose behavior we can understand and directly track to specific architectural characteristics. Like many scientific applications, I/O is essentially nonexistent in this workload.

The two applications that we use in this appendix are Barnes and Ocean, which represent two important but very different types of parallel computation. We briefly describe each of these applications and kernels and characterize their basic behavior in terms of parallelism and communication. We describe how the problem is decomposed for a distributed shared-memory multiprocessor; certain data decompositions that we describe are not necessary on multiprocessors that have a single, centralized memory.

The FFT Kernel

The FFT is the key kernel in applications that use spectral methods, which arise in fields ranging from signal processing to fluid flow to climate modeling. The FFT application we study here is a one-dimensional version of a parallel algorithm for a complex number FFT. It has a sequential execution time for n data points of $n \log n$. The algorithm uses a high radix (equal to $\sqrt{n}$) that minimizes communication. The measurements shown in this appendix are collected for a million-point input data set.

There are three primary data structures: the input and output arrays of the data being transformed and the roots of unity matrix, which is precomputed and only read during the execution. All arrays are organized as square matrices. The six steps in the algorithm are as follows:

1. Transpose data matrix.
2. Perform 1D FFT on each row of data matrix.
3. Multiply the roots of unity matrix by the data matrix and write the result in the data matrix.
4. Transpose data matrix.
5. Perform 1D FFT on each row of data matrix.
6. Transpose data matrix.

The data matrices and the roots of unity matrix are partitioned among processors in contiguous chunks of rows, so that each processor's partition falls in its own local memory. The first row of the roots of unity matrix is accessed heavily by all processors and is often replicated, as we do, during the first step of the algorithm just shown. The data transposes ensure good locality during the individual FFT steps, which would otherwise access nonlocal data.

The only communication is in the transpose phases, which require all-to-all communication of large amounts of data. Contiguous subcolumns in the rows assigned to a processor are grouped into blocks, which are transposed and placed into the proper location of the destination matrix. Every processor transposes one block locally and sends one block to each of the other processors in the system. Although there is no reuse of individual words in the transpose, with long cache blocks it makes sense to block the transpose to take advantage of the spatial locality afforded by long blocks in the source matrix.

The LU Kernel

LU is an LU factorization of a dense matrix and is representative of many dense linear algebra computations, such as QR factorization, Cholesky factorization, and eigenvalue methods. For a matrix of size $n \times n$ the running time is n^3 and the parallelism is proportional to n^2. Dense LU factorization can be performed efficiently by blocking the algorithm, using the techniques in Chapter 2, which leads to highly efficient cache behavior and low communication. After blocking the algorithm, the dominant computation is a dense matrix multiply that occurs in the innermost loop. The block size is chosen to be small enough to keep the cache miss rate low and large enough to reduce the time spent in the less parallel parts of the computation. Relatively small block sizes (8 × 8 or 16 × 16) tend to satisfy both criteria.

Two details are important for reducing interprocessor communication. First, the blocks of the matrix are assigned to processors using a 2D tiling: The $\frac{n}{B} \times \frac{n}{B}$ (where each block is $B \times B$) matrix of blocks is allocated by laying a grid of size $p \times p$ over the matrix of blocks in a cookie-cutter fashion until all the blocks are allocated to a processor. Second, the dense matrix multiplication is performed by the processor that owns the *destination* block. With this blocking and allocation scheme, communication during the reduction is both regular and predictable. For the measurements in this appendix, the input is a 512 × 512 matrix and a block of 16 × 16 is used.

A natural way to code the blocked LU factorization of a 2D matrix in a shared address space is to use a 2D array to represent the matrix. Because blocks are allocated in a tiled decomposition, and a block is not contiguous in the address space in a 2D array, it is very difficult to allocate blocks in the local memories of the processors that own them. The solution is to ensure that blocks assigned to a processor are allocated locally and contiguously by using a 4D array (with the first two dimensions specifying the block number in the 2D grid of blocks, and the next two specifying the element in the block).

The Barnes Application

Barnes is an implementation of the Barnes-Hut n-body algorithm solving a problem in galaxy evolution. *N-body algorithms* simulate the interaction among a large number of bodies that have forces interacting among them. In this instance, the bodies represent collections of stars and the force is gravity. To reduce the computational time required to model completely all the individual interactions among the bodies, which grow as n^2, n-body algorithms take advantage of the fact that the forces drop off with distance. (Gravity, for example, drops off as $1/d^2$, where d is the distance between the two bodies.) The Barnes-Hut algorithm takes advantage of this property by treating a collection of bodies that are “far away” from another body as a single point at the center of mass of the collection and with mass equal to the collection. If the body is far enough from any body in the collection, then the error introduced will be negligible. The collections are structured in a hierarchical fashion, which can be represented in a tree. This algorithm yields an $n \log n$ running time with parallelism proportional to n.

The Barnes-Hut algorithm uses an octree (each node has up to eight children) to represent the eight cubes in a portion of space. Each node then represents the collection of bodies in the subtree rooted at that node, which we call a *cell*. Because the density of space varies and the leaves represent individual bodies, the depth of the tree varies. The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm for a body starts at the root of the tree. For every node in the tree it visits, the algorithm determines if the center of mass of the cell represented by the subtree rooted at the node is "far enough away" from the body. If so, the entire subtree under that node is approximated by a single point at the center of mass of the cell, and the force that this center of mass exerts on the body is computed. On the other hand, if the center of mass is not far enough away, the cell must be "opened" and each of its subtrees visited. The distance between the body and the cell, together with the error tolerances, determines which cells must be opened. This force calculation phase dominates the execution time. This appendix takes measurements using 16K bodies; the criterion for determining whether a cell needs to be opened is set to the middle of the range typically used in practice.

Obtaining effective parallel performance on Barnes-Hut is challenging because the distribution of bodies is nonuniform and changes over time, making partitioning the work among the processors and maintenance of good locality of reference difficult. We are helped by two properties: (1) the system evolves slowly, and (2) because gravitational forces fall off quickly, with high probability, each cell requires touching a small number of other cells, most of which were used on the last time step. The tree can be partitioned by allocating each processor a subtree. Many of the accesses needed to compute the force on a body in the subtree will be to other bodies in the subtree. Since the amount of work associated with a subtree varies (cells in dense portions of space will need to access more cells), the size of the subtree allocated to a processor is based on some measure of the work it has to do (e.g., how many other cells it needs to visit), rather than just on the number of nodes in the subtree. By partitioning the octree representation, we can obtain good load balance and good locality of reference, while keeping the partitioning cost low. Although this partitioning scheme results in good locality of reference, the resulting data references tend to be for small amounts of data and are unstructured. Thus, this scheme requires an efficient implementation of shared-memory communication.

The Ocean Application

Ocean simulates the influence of eddy and boundary currents on large-scale flow in the ocean. It uses a restricted red-black Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations. *Red-black Gauss-Seidel* is an iteration technique that colors the points in the grid so as to consistently update each point based on previous values of the adjacent neighbors. *Multigrid methods* solve finite difference equations by iteration using hierarchical grids. Each grid in the hierarchy has fewer points than the grid below and is an approximation to the lower grid. A finer grid increases accuracy and thus the rate of convergence, while requiring more execution time, since it has more data points. Whether to move up or down in the hierarchy of grids used for the next iteration is determined by the rate of change of the data values. The estimate of the error at every time step is used to decide whether to stay at the same grid, move to a coarser grid, or move to a finer grid. When the iteration converges at the finest level, a solution has been reached. Each iteration has n^2 work for an $n \times n$ grid and the same amount of parallelism.

The arrays representing each grid are dynamically allocated and sized to the particular problem. The entire ocean basin is partitioned into square subgrids (as close as possible) that are allocated in the portion of the address space corresponding to the local memory of the individual processors, which are assigned responsibility for the subgrid. For the measurements in this appendix we use an input that has 130 × 130 grid points. There are five steps in a time iteration. Since data are exchanged between the steps, all the processors present synchronize at the end of each step before proceeding to the next. Communication occurs when the boundary points of a subgrid are accessed by the adjacent subgrid in nearest-neighbor fashion.

Computation/Communication for the Parallel Programs

A key characteristic in determining the performance of parallel programs is the ratio of computation to communication. If the ratio is high, it means the application has lots of computation for each datum communicated. As we saw in Section I.2, communication is the costly part of parallel computing; therefore, high computation-to-communication ratios are very beneficial. In a parallel processing environment, we are concerned with how the ratio of computation to communication changes as we increase either the number of processors, the size of the problem, or both. Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up. Because we are often interested in running larger problems, it is vital to understand how changing the data set size affects this ratio.

To understand what happens quantitatively to the computation-to-communication ratio as we add processors, consider what happens separately to computation and to communication as we either add processors or increase problem size. Figure I.1 shows that as we add processors, for these applications, the amount of computation per processor falls proportionately and the amount of communication per processor falls more slowly. As we increase the problem size, the computation scales as the $O(\ )$ complexity of the algorithm dictates. Communication scaling is more complex and depends on details of the algorithm; we describe the basic phenomena for each application in the caption of Figure I.1.

Application	Scaling of computation	Scaling of communication	Scaling of computation-to-communication
FFT	$\frac{n\log n}{p}$	$\frac{n}{p}$	$\log n$
LU	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$
Barnes	$\frac{n\log n}{p}$	approximately $\frac{\sqrt{n}(\log n)}{\sqrt{p}}$	approximately $\frac{\sqrt{n}}{\sqrt{p}}$
Ocean	$\frac{n}{p}$	$\frac{\sqrt{n}}{\sqrt{p}}$	$\frac{\sqrt{n}}{\sqrt{p}}$

Figure I.1 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel multiprocessors. In this table, *p* is the increased processor count and *n* is the increased dataset size. Scaling is on a per-processor basis. The computation scales up with *n* at the rate given by *O*( ) analysis and scales down linearly as *p* is increased. Communication scaling is more complex. In FFT, all data points must interact, so communication increases with *n* and decreases with *p*. In LU and Ocean, communication is proportional to the boundary of a block, so it scales with dataset size at a rate proportional to the side of a square with *n* points, namely, $\sqrt{n}$; for the same reason communication in these two applications scales inversely to $\sqrt{p}$. Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies that require communication scales as $\sqrt{n}$. An additional factor of log *n* is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to $\sqrt{p}$.

The overall computation-to-communication ratio is computed from the individual growth rate in computation and communication. In general, this ratio rises slowly with an increase in dataset size and decreases as we add processors. This reminds us that performing a fixed-size problem with more processors leads to increasing inefficiencies because the amount of communication among processors grows. It also tells us how quickly we must scale dataset size as we add processors to keep the fraction of time in communication fixed. The following example illustrates these trade-offs.

Example Suppose we know that for a given multiprocessor the Ocean application spends 20% of its execution time waiting for communication when run on four processors. Assume that the cost of each communication event is independent of processor count, which is not true in general, since communication costs rise with processor count. How much faster might we expect Ocean to run on a 32-processor machine with the same problem size? What fraction of the execution time is spent on communication in this case? How much larger a problem should we run if we want the fraction of time spent communicating to be the same?

Answer The computation-to-communication ratio for Ocean is $\sqrt{n}/\sqrt{p}$, so if the problem size is the same, the communication frequency scales by $\sqrt{p}$. This means that communication time increases by $\sqrt{8}$. We can use a variation on Amdahl's law, recognizing that the computation is decreased but the communication time is increased. If T_4 is the total execution time for four processors, then the execution time for 32 processors is

$$\begin{aligned} T_{32} &= \text{Compute time} + \text{Communication time} \\ &= \frac{0.8 \times T_4}{8} + (0.2 \times T_4) \times \sqrt{8} \\ &= 0.1 \times T_4 + 0.57 \times T_4 = 0.67 \times T_4 \end{aligned}$$

Hence, the speedup is

$$\text{Speedup} = \frac{T_4}{T_{32}} = \frac{T_4}{0.67 \times T_4} = 1.49$$

and the fraction of time spent in communication goes from 20% to $0.57/0.67 = 85\%$.

For the fraction of the communication time to remain the same, we must keep the computation-to-communication ratio the same, so the problem size must scale at the same rate as the processor count. Notice that, because we have changed the problem size, we cannot fairly compare the speedup of the original problem and the scaled problem. We will return to the critical issue of scaling applications for multiprocessors in Section I.6.

I.4 Synchronization: Scaling Up

In this section, we focus first on synchronization performance problems in larger multiprocessors and then on solutions for those problems.

Synchronization Performance Challenges

To understand why the simple spin lock scheme presented in Chapter 5 does not scale well, imagine a large multiprocessor with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following example shows how bad things can be.

Example Suppose there are 10 processors on a bus and each tries to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 100 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 10 requests? Assume that the bus is totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

Answer When i processes are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

i load linked operations to access the lock

i store conditional operations to try to lock the lock

1 store (to release the lock)

Thus, for i processes, there are a total of $2i+1$ bus transactions. Note that this assumes that the critical section time is negligible, so that the lock is released before any other processors whose store conditional failed attempt another load linked.

Thus, for n processes, the total number of bus operations is

$$\sum_{i=1}^{n}(2i+1) = n(n+1)+n = n^2+2n$$

For 10 processes there are 120 bus transactions requiring 12,000 clock cycles or 120 clock cycles per lock acquisition!

The difficulty in this example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. (The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist.) The key advantages of spin locks—that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor—are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one to protect a counter that tallies the processes arriving at the barrier and one to hold the processes until the last process arrives at the barrier. To implement a barrier, we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure I.2 is a typical implementation, assuming that `lock` and `unlock` provide basic spin locks and `total` is the number of processes that must reach the barrier.

```
lock (counterlock);/* ensure update atomic */
if (count==0) release=0;/* first=>reset release */
count = count + 1;/* count arrivals */
unlock(counterlock);/* release lock */
if (count==total) {/* all arrived */
      count=0;/* reset counter */
      release=1;/* release processes */
}
else {/* more to come */
      spin (release==1);/* wait for arrivals */
}
```

Figure I.2 Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier. The operation `spin (release==1)` causes a process to wait until all processes reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The "fast" process then traps the remaining "slow" process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier because one process is trapped at the last instance, and the number of processes can never reach the value of total.

The important observation in this example is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This extra step would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier*, which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure I.3 shows the code for the sense-reversing barrier. This version of a barrier is safely usable; as the next example shows, however, its performance can still be quite poor.

```
local_sense =! local_sense; /* toggle local_sense */
lock (counterlock);/* ensure update atomic */
count=count+1;/* count arrivals */
if (count==total) {/* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
unlock (counterlock);/* unlock */
spin (release==local_sense);/* wait for signal */
}
```

Figure I.3 Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag `release`, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure I.2.

Example Suppose there are 10 processors on a bus and each tries to execute a barrier simultaneously. Assume that each bus transaction is 100 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 10 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

Answer We assume that load linked and store conditional are used to implement lock and unlock. Figure I.4 shows the sequence of bus events for a processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock. There is a slight difference for the last process to reach the barrier, as described in the caption.

For the ith process, the number of bus transactions is $3i+4$. The last process to reach the barrier requires one less. Thus, for n processes, the number of bus transactions is

$$\left(\sum_{i=1}^{n}(3i+4)\right)-1=\frac{3n^2+11n}{2}-1$$

For 10 processes, this is 204 bus cycles or 20,400 clock cycles! Our barrier operation takes almost twice as long as the 10-processor lock-unlock sequence.

Event	Number of times for process *i*	Corresponding source line	Comment
LL `counterlock`	i	`lock (counterlock);`	All processes try for lock.
Store conditional	i	`lock (counterlock);`	All processes try for lock.
LD `count`	1	`count = count + 1;`	Successful process.
Load linked	$i-1$	`lock (counterlock);`	Unsuccessful process; try again.
SD `count`	1	`count = count + 1;`	Miss to get exclusive access.
SD `counterlock`	1	`unlock(counterlock);`	Miss to get the lock.
LD `release`	2	`spin (release==local_sense);/`	Read release: misses initially and when finally written.

Figure I.4 Here are the actions, which require a bus transaction, taken when the *i*th process reaches the barrier. The last process to reach the barrier requires one less bus transaction, since its read of release for the spin will hit in the cache!

As we can see from these examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these examples is the serialization of each process's attempt to complete the synchronization. This serialization is a problem when there is contention because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 10 lock and unlock operations depended only on the latency in the uncontended case, then it would take 1000 rather than 15,000 cycles to complete the synchronization operations. The barrier situation is as bad, and in some ways worse, since it is highly likely to incur contention. The use of a bus interconnect exacerbates these problems, but serialization could be just as serious in a directory-based multiprocessor, where the latency would be large. The next subsection presents some solutions that are useful when either the contention is high or the processor count is large.

Synchronization Mechanisms for Larger-Scale Multiprocessors

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

Software Implementations

The major difficulty with our spin lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure I.5 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see Sections F.4 to F.8). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer

```
         DADDUI  R3,R0,#1    ;R3 = initial delay
lockit:  LL      R2,0(R1)    ;load linked
         BNEZ    R2,lockit   ;not available-spin
         DADDUI  R2,R2,#1    ;get locked value
         SC      R2,0(R1)    ;store conditional
         BNEZ    R2,gotit    ;branch if store succeeds
         DSLL    R3,R3,#1    ;increase delay by factor of 2
         PAUSE   R3          ;delays by value in R3
         J       lockit
gotit:   use data protected by lock
```

Figure I.5 A spin lock with exponential back-off. When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing a copy of the value in R3 until it reaches 0. The exact timing of the delay is multiprocessor dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement `pause R3` should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again. The small variations in the rate at which competing processors execute instructions are usually sufficient to ensure that processes will not continually collide. If the natural perturbation in execution time was insufficient, R3 could be initialized with a small random value, increasing the variance in the successive delays and reducing the probability of successive collisions.

performance when the lock was in use by only two processes and the first one happened to find it locked.

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits. Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree*, a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there.

Our combining tree barrier uses a predetermined n-ary tree structure. We use the variable k to stand for the fan-in; in practice, $k=4$ seems to work well. When the kth process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. A tree-based barrier, as shown in Figure I.6, uses a tree to combine the processes and a single signal to release the barrier. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but more recent machines have relied on software libraries for this support.

Hardware Primitives

In this subsection, we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases, we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 10 lock/unlock sequences, as we saw in the example on page I-12.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, which we

```
struct node{/* a node in the combining tree */
       int counterlock; /* lock for this node */
       int count; /* counter for this node */
       int parent; /* parent in the tree = 0..P-1 except for root */
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode, int local_sense) {
       lock (tree[mynode].counterlock); /* protect count */
       tree[mynode].count=tree[mynode].count+1;
               /* increment count */
       if (tree[mynode].count==k) {/* all arrived at mynode */
               if (tree[mynode].parent >=0) {
                       barrier(tree[mynode].parent);
               } else{
                       release = local_sense;
               };
               tree[mynode].count = 0; /* reset for the next time */
       unlock (tree[mynode].counterlock); /* unlock */
       spin (release==local_sense); /* wait */
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

Figure I.6 An implementation of a tree-based barrier reduces contention considerably. The tree is assumed to be prebuilt statically using the nodes in the array tree. Each node in the tree combines *k* processes and provides a separate counter and lock, so that at most *k* processes contend at each node. When the *k*th process reaches a node in the tree, it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches *k*, the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. The value of `tree[root].parent` should be set to −1 when the tree is initially built.

describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based multiprocessor where the individual processor caches are addressable. In a bus-based multiprocessor, a software implementation would be more appropriate and would have each processor using a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable, the controller creates a record of the node's request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

Example How many bus transactions and how long does it take to have 10 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as those in the earlier example on page I-12.

Answer For n processors, each will initially attempt a lock access, generating a bus transaction; one will succeed and free up the lock, for a total of $n+1$ transactions for the first processor. Each subsequent processor requires two bus transactions, one to receive the lock and one to free it up. Thus, the total number of bus transactions is $(n+1)+2(n-1)=3n-1$. Note that the number of bus transactions is now linear in the number of processors contending for the lock, rather than quadratic, as it was with the spin lock we examined earlier. For 10 processors, this requires 29 bus cycles or 2900 clock cycles.

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based multiprocessor, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation. Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment*, which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using fetch-and-increment we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

Example Write the code for the barrier using fetch-and-increment. Making the same assumptions as in our earlier example and also assuming that a fetch-and-increment operation, which returns the incremented value, takes 100 clock cycles, determine the time for 10 processors to traverse the barrier. How many bus cycles are required?

```
local_sense =! local_sense; /* toggle local_sense */
fetch_and_increment(count);/* atomic update */
if (count==total) {/* all arrived */
      count=0;/* reset counter */
      release=local_sense;/* release processes */
}
else {/* more to come */
      spin (release==local_sense);/* wait for signal */
}
```

Figure I.7 Code for a sense-reversing barrier using fetch-and-increment to do the counting.

Answer Figure I.7 shows the code for the barrier. For *n* processors, this implementation requires *n* fetch-and-increment operations, *n* cache misses to access the count, and *n* cache misses for the release operation, for a total of 3*n* bus transactions. For 10 processors, this is 30 bus transactions or 3000 clock cycles. Like the queueing lock, the time is linear in the number of processors. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.

As we have seen, synchronization problems can become quite acute in largerscale multiprocessors. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel processors is very challenging.

I.5 Performance of Scientific Applications on Shared-Memory Multiprocessors

This section covers the performance of the scientific applications from Section I.3 on both symmetric shared-memory and distributed shared-memory multiprocessors.

Performance of a Scientific Workload on a Symmetric Shared-Memory Multiprocessor

We evaluate the performance of our four scientific applications on a symmetric shared-memory multiprocessor using the following problem sizes:

- *Barnes-Hut*—16 K bodies run for six time steps (the accuracy control is set to 1.0, a typical, realistic value)

- *FFT*—1 million complex data points
- *LU*—A 512 × 512 matrix is used with 16 × 16 blocks
- *Ocean*—A 130 × 130 grid with a typical error tolerance

In looking at the miss rates as we vary processor count, cache size, and block size, we decompose the total miss rate into *coherence misses* and normal uniprocessor misses. The normal uniprocessor misses consist of capacity, conflict, and compulsory misses. We label these misses as capacity misses because that is the dominant cause for these benchmarks. For these measurements, we include as a coherence miss any write misses needed to upgrade a block from shared to exclusive, even though no one is sharing the cache block. This measurement reflects a protocol that does not distinguish between a private and shared cache block.

Figure I.8 shows the data miss rates for our four applications, as we increase the number of processors from 1 to 16, while keeping the problem size constant. As we increase the number of processors, the total amount of cache increases, usually causing the capacity misses to drop. In contrast, increasing the processor count usually causes the amount of communication to increase, in turn causing the coherence misses to rise. The magnitude of these two effects differs by application.

In FFT, the capacity miss rate drops (from nearly 7% to just over 5%) but the coherence miss rate increases (from about 1% to about 2.7%), leading to a constant overall miss rate. Ocean shows a combination of effects, including some that relate to the partitioning of the grid and how grid boundaries map to cache blocks. For a typical 2D grid code the communication-generated misses are proportional to the boundary of each partition of the grid, while the capacity misses are proportional to the area of the grid. Therefore, increasing the total amount of cache while keeping the total problem size fixed will have a more significant effect on the capacity miss rate, at least until each subgrid fits within an individual processor's cache. The significant jump in miss rate between one and two processors occurs because of conflicts that arise from the way in which the multiple grids are mapped to the caches. This conflict is present for direct-mapped and two-way set associative caches, but fades at higher associativities. Such conflicts are not unusual in array-based applications, especially when there are multiple grids in use at once. In Barnes and LU, the increase in processor count has little effect on the miss rate, sometimes causing a slight increase and sometimes causing a slight decrease.

Increasing the cache size usually has a beneficial effect on performance, since it reduces the frequency of costly cache misses. Figure I.9 illustrates the change in miss rate as cache size is increased for 16 processors, showing the portion of the miss rate due to coherence misses and to uniprocessor capacity misses. Two effects can lead to a miss rate that does not decrease—at least not as quickly as we might expect—as cache size increases: inherent communication and plateaus in the miss rate. Inherent communication leads to a certain frequency of coherence misses that are not significantly affected by increasing cache size. Thus, if the cache size is increased while maintaining a fixed problem size, the coherence miss rate

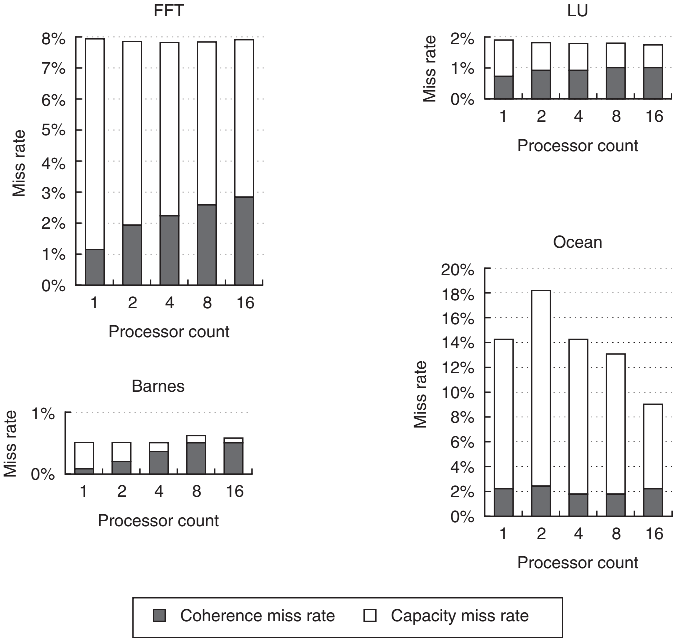


Figure I.8 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16. The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that are potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data are potentially shared if they are allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data are heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that are potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single-processor numbers include write upgrade misses, which occur in this protocol even if the data are not actually shared, since they are in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the *y*-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.

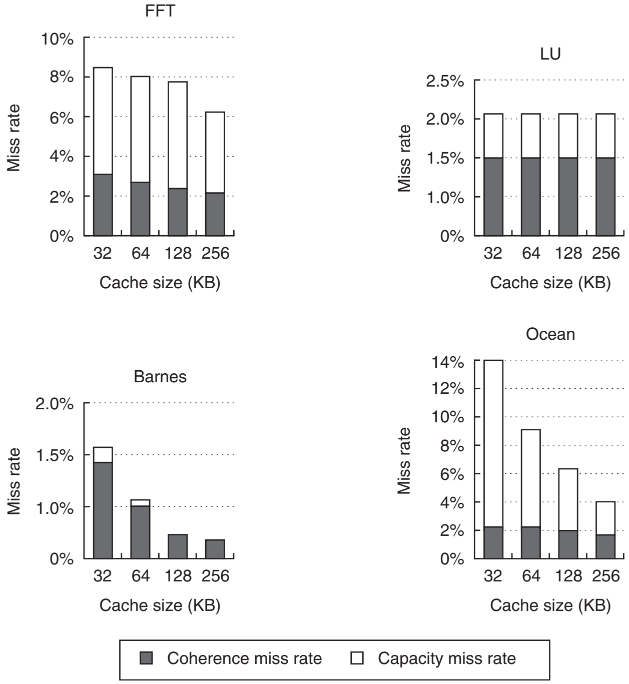


Figure I.9 The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

eventually limits the decrease in cache miss rate. This effect is most obvious in Barnes, where the coherence miss rate essentially becomes the entire miss rate.

A less important effect is a temporary plateau in the capacity miss rate that arises when the application has some fraction of its data present in cache but some significant portion of the dataset does not fit in the cache or in caches that are slightly bigger. In LU, a very small cache (about 4 KB) can capture the pair of 16 × 16 blocks used in the inner loop; beyond that, the next big improvement in capacity miss rate occurs when both matrices fit in the caches, which occurs when the total cache size is between 4 MB and 8 MB. This effect, sometimes called a *working set effect*, is partly at work between 32 KB and 128 KB for FFT, where the capacity miss rate drops only 0.3%. Beyond that cache size, a faster decrease in the capacity miss rate is seen, as a major data structure begins to reside in the cache. These plateaus are common in programs that deal with large arrays in a structured fashion.

Increasing the block size is another way to change the miss rate in a cache. In uniprocessors, larger block sizes are often optimal with larger caches. In multiprocessors, two new effects come into play: a reduction in spatial locality for shared data and a potential increase in miss rate due to false sharing. Several studies have shown that shared data have lower spatial locality than unshared data. Poorer locality means that, for shared data, fetching larger blocks is less effective than in a uniprocessor because the probability is higher that the block will be replaced before all its contents are referenced. Likewise, increasing the basic size also increases the potential frequency of false sharing, increasing the miss rate.

Figure I.10 shows the miss rates as the cache block size is increased for a 16-processor run with a 64 KB cache. The most interesting behavior is in Barnes, where the miss rate initially declines and then rises due to an increase in the number of coherence misses, which probably occurs because of false sharing. In the other benchmarks, increasing the block size decreases the overall miss rate. In Ocean and LU, the block size increase affects both the coherence and capacity miss rates about equally. In FFT, the coherence miss rate is actually decreased at a faster rate than the capacity miss rate. This reduction occurs because the communication in FFT is structured to be very efficient. In less optimized programs, we would expect more

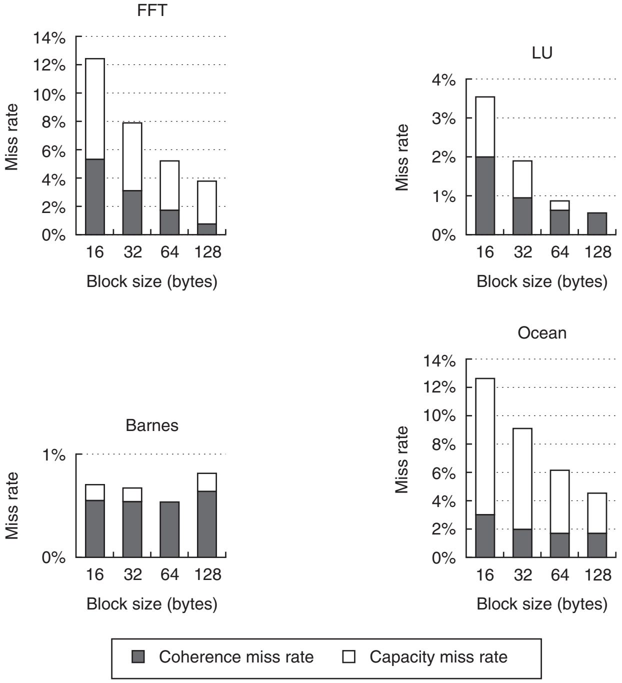


Figure I.10 The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64 KB cache and two-way set associativity. Once again we use different scales for each benchmark.

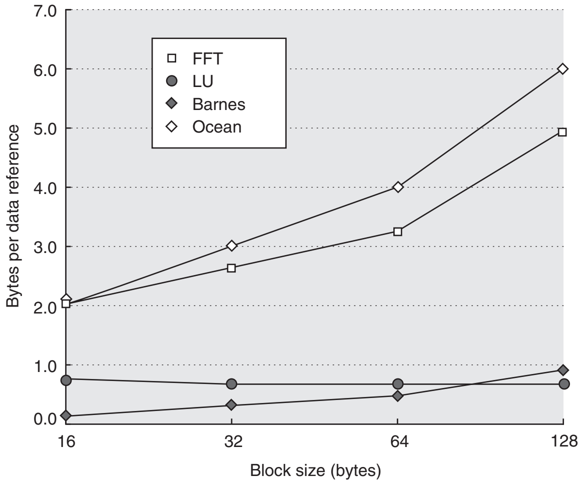


Figure I.11 Bus traffic for data misses climbs steadily as the block size in the data cache is increased. The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership or upgrade misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT, this simplification accounts for less than 10% of the traffic.

false sharing and less spatial locality for shared data, resulting in more behavior like that of Barnes.

Although the drop in miss rates with longer blocks may lead you to believe that choosing a longer block size is the best decision, the bottleneck in bus-based multiprocessors is often the limited memory and bus bandwidth. Larger blocks mean more bytes on the bus per miss. Figure I.11 shows the growth in bus traffic as the block size is increased. This growth is most serious in the programs that have a high miss rate, especially Ocean. The growth in traffic can actually lead to performance slowdowns due both to longer miss penalties and to increased bus contention.

Performance of a Scientific Workload on a Distributed-Memory Multiprocessor

The performance of a directory-based multiprocessor depends on many of the same factors that influence the performance of bus-based multiprocessors (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our scientific/technical workload and then look at the effect of different types of misses.

Because the multiprocessor is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a larger block size of 64 bytes.

In distributed-memory architectures, the distribution of memory requests between local and remote is key to performance because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section, we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution such misses can be significant.

As Figure I.12 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise results from two factors: an increase in mapping conflicts in the cache that occur when the grid becomes small, which leads to a rise in local misses, and an increase in the number of the coherence misses, which are all remote.

Figure I.13 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would amplify this trend.

We examine the effect of changing the block size in Figure I.14. Because these applications have good spatial locality, increases in block size reduce the miss rate, even for large blocks, although the performance benefits for going to the largest blocks are small. Furthermore, most of the improvement in miss rate comes from a reduction in the local misses.

Rather than plot the memory traffic, Figure I.15 plots the number of bytes required per data reference versus block size, breaking the requirement into local and global bandwidth. In the case of a bus, we can simply aggregate the demands of each processor to find the total demand for bus and memory bandwidth. For a scalable interconnect, we can use the data in Figure I.15 to compute the required per-node global bandwidth and the estimated bisection bandwidth, as the next example shows.

Example Assume a 64-processor multiprocessor with 1 GHz processors that sustain one memory reference per processor clock. For a 64-byte block size, the remote miss rate is 0.7%. Find the per-node and estimated bisection bandwidth for FFT. Assume that the processor does not stall for remote memory requests; this might be true if, for example, all remote data were prefetched. How do these bandwidth requirements compare to various interconnection technologies?

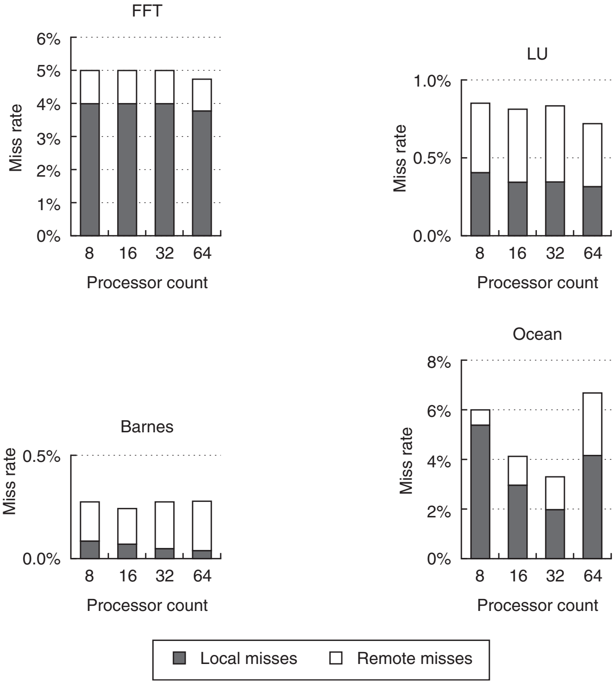


Figure I.12 The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that, to show the detailed behavior of each benchmark, different scales are used on the *y*-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data are up to date in the local memory but cached elsewhere and, therefore, require invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model.

FFT performs all-to-all communication, so the bisection bandwidth is equal to the number of processors times the per-node bandwidth, or about 64 × 448 MB/sec = 28.7 GB/sec. The SGI Origin 3000 with 64 processors has a bisection bandwidth of about 50 GB/sec. No standard networking technology comes close.

Answer The per-node bandwidth is simply the number of data bytes per reference times the reference rate: 0.7% × 1 GB/sec × 64 = 448 MB/sec. This rate is somewhat higher than the hardware sustainable transfer rate for the CrayT3E (using a block prefetch)

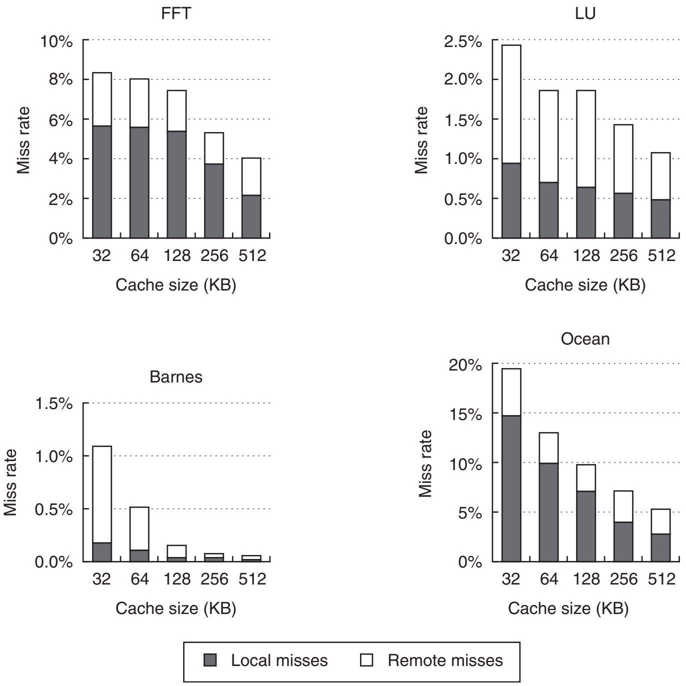


Figure I.13 Miss rates decrease as cache sizes grow. Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

and lower than that for an SGI Origin 3000 (1.6 GB/processor pair). The FFT per-node bandwidth demand exceeds the bandwidth sustainable from the fastest standard networks by more than a factor of 5.

The previous example looked at the bandwidth demands. The other key issue for a parallel program is remote memory access time, or latency. To get insight into this, we use a simple example of a directory-based multiprocessor. Figure I.16 shows the parameters we assume for our simple multiprocessor model. It assumes that the time to first word for a local memory access is 85 processor cycles and that the path to local memory is 16 bytes wide, while the network interconnect is 4 bytes wide. This model ignores the effects of contention, which are probably not too serious in the parallel benchmarks we examine, with the possible exception of FFT, which uses all-to-all communication. Contention could have a serious performance impact in other workloads.

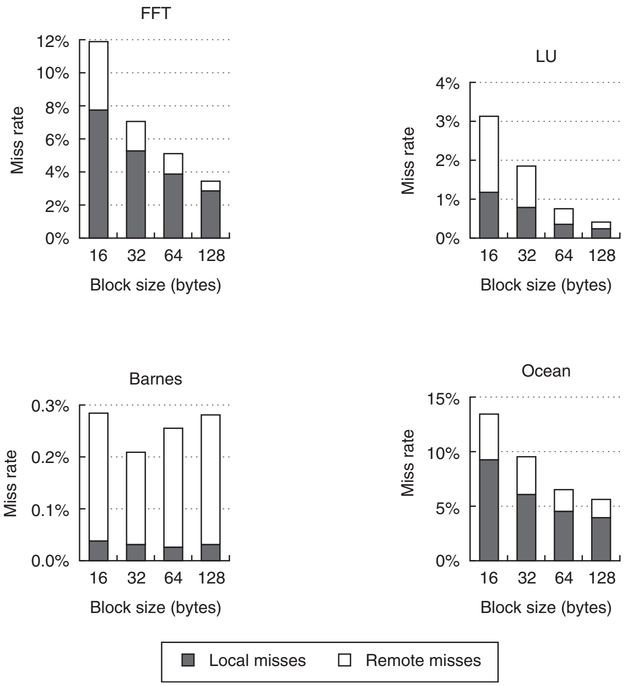


Figure I.14 Data miss rate versus block size assuming a 128 KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

Figure I.17 shows the cost in cycles for the average memory reference, assuming the parameters in Figure I.16. Only the latencies for each reference type are counted. Each bar indicates the contribution from cache hits, local misses, remote misses, and three-hop remote misses. The cost is influenced by the total frequency of cache misses and upgrades, as well as by the distribution of the location where the miss is satisfied. The cost for a remote memory reference is fairly steady as the processor count is increased, except for Ocean. The increasing miss rate in Ocean for 64 processors is clear in Figure I.12. As the miss rate increases, we should expect the time spent on memory references to increase also.

Although Figure I.17 shows the memory access cost, which is the dominant multiprocessor cost in these benchmarks, a complete performance model would need to consider the effect of contention in the memory system, as well as the losses arising from synchronization delays.

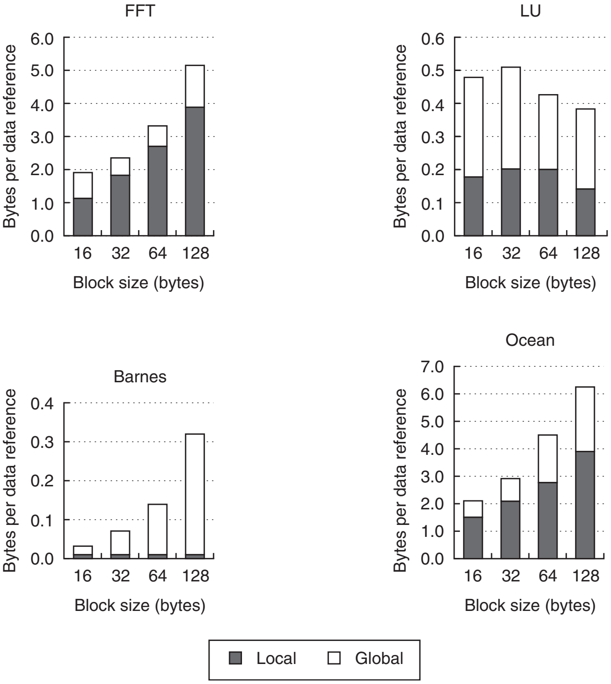


Figure I.15 The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assume a 128 KB cache for each of 64 processors.

Characteristic	**Processor clock cycles ≤16 processors**	**Processor clock cycles 17–64 processors**
Cache hit	1	1
Cache miss to local memory	85	85
Cache miss to remote home directory	125	150
Cache miss to remotely cached data (three-hop miss)	140	170

Figure I.16 Characteristics of the example directory-based multiprocessor. Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a three-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time but does include some factor for increasing interconnect time. These remote access latencies are based on those in an SGI Origin 3000, the fastest scalable interconnect system in 2001, and assume a 500 MHz processor.

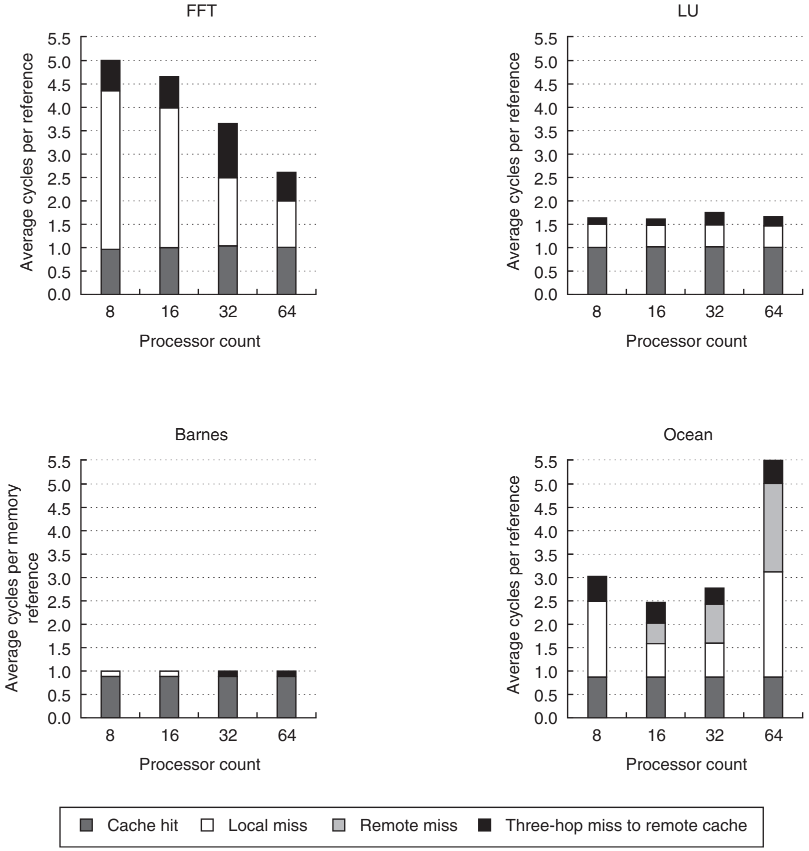


Figure I.17 The effective latency of memory references in a DSM multiprocessor depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 2) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 512 KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure I.16. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (1–4%) and a significant three-hop miss rate (1%). The improvement in FFT comes from the reduction in local miss rate from 4% to 1%, as the aggregate cache increases. Ocean shows the biggest change in the cost of memory accesses, and the highest overall cost at 64 processors. The high cost is driven primarily by a high local miss rate (average 1.6%). The memory access cost drops from 8 to 16 processors as the grids more easily fit in the individual caches. At 64 processors, the dataset size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure I.12.

I.6 Performance Measurement of Parallel Processors with Scientific Applications

One of the most controversial issues in parallel processing has been how to measure the performance of parallel processors. Of course, the straightforward answer is to measure a benchmark as supplied and to examine wall-clock time. Measuring wall-clock time obviously makes sense; in a parallel processor, measuring CPU time can be misleading because the processors may be idle but unavailable for other uses.

Users and designers are often interested in knowing not just how well a multiprocessor performs with a certain fixed number of processors, but also how the performance scales as more processors are added. In many cases, it makes sense to scale the application or benchmark, since if the benchmark is unscaled, effects arising from limited parallelism and increases in communication can lead to results that are pessimistic when the expectation is that more processors will be used to solve larger problems. Thus, it is often useful to measure the speedup as processors are added both for a fixed-size problem and for a scaled version of the problem, providing an unscaled and a scaled version of the speedup curves. The choice of how to measure the uniprocessor algorithm is also important to avoid anomalous results, since using the parallel version of the benchmark may understate the uniprocessor performance and thus overstate the speedup.

Once we have decided to measure scaled speedup, the question is *how* to scale the application. Let's assume that we have determined that running a benchmark of size n on p processors makes sense. The question is how to scale the benchmark to run on $m \times p$ processors. There are two obvious ways to scale the problem: (1) keeping the amount of memory used per processor constant, and (2) keeping the total execution time, assuming perfect speedup, constant. The first method, called *memory-constrained scaling*, specifies running a problem of size $m \times n$ on $m \times p$ processors. The second method, called *time-constrained scaling*, requires that we know the relationship between the running time and the problem size, since the former is kept constant. For example, suppose the running time of the application with data size n on p processors is proportional to n^2/p. Then, with time-constrained scaling, the problem to run is the problem whose ideal running time on $m \times p$ processors is still n^2/p. The problem with this ideal running time has size $\sqrt{m} \times n$.

Example Suppose we have a problem whose execution time for a problem of size n is proportional to n^3. Suppose the actual running time on a 10-processor multiprocessor is 1 hour. Under the time-constrained and memory-constrained scaling models, find the size of the problem to run and the effective running time for a 100-processor multiprocessor.

Answer For the time-constrained problem, the ideal running time is the same, 1 hour, so the problem size is $\sqrt[3]{10} \times \mathrm{n}$ or 2.15 times larger than the original. For memory-constrained scaling, the size of the problem is $10n$ and the ideal execution time is $10^3/10$, or 100 hours! Since most users will be reluctant to run a problem on an order of magnitude more processors for 100 times longer, this size problem is probably unrealistic.

In addition to the scaling methodology, there are questions as to how the program should be scaled when increasing the problem size affects the quality of the result. Often, we must change other application parameters to deal with this effect. As a simple example, consider the effect of time to convergence for solving a differential equation. This time typically increases as the problem size increases, since, for example, we often require more iterations for the larger problem. Thus, when we increase the problem size, the total running time may scale faster than the basic algorithmic scaling would indicate.

For example, suppose that the number of iterations grows as the log of the problem size. Then, for a problem whose algorithmic running time is linear in the size of the problem, the effective running time actually grows proportional to $n \log n$. If we scaled from a problem of size m on 10 processors, purely algorithmic scaling would allow us to run a problem of size $10\,m$ on 100 processors. Accounting for the increase in iterations means that a problem of size $k \times m$, where $k \log k = 10$, will have the same running time on 100 processors. This problem size yields a scaling of $5.72\,m$, rather than $10\,m$.

In practice, scaling to deal with error requires a good understanding of the application and may involve other factors, such as error tolerances (for example, it affects the cell-opening criteria in Barnes-Hut). In turn, such effects often significantly affect the communication or parallelism properties of the application as well as the choice of problem size.

Scaled speedup is not the same as unscaled (or true) speedup; confusing the two has led to erroneous claims (e.g., see the discussion in Section I.6). Scaled speedup has an important role, but only when the scaling methodology is sound and the results are clearly reported as using a scaled version of the application. Singh, Hennessy, and Gupta [1993] described these issues in detail.

I.7 Implementing Cache Coherence

In this section, we explore the challenge of implementing cache coherence, starting first by dealing with the challenges in a snooping coherence protocol, which we simply alluded to in Chapter 5. Implementing a directory protocol adds some additional complexity to a snooping protocol, primarily arising from the absence of broadcast, which forces the use of a different mechanism to resolve races. Furthermore, the larger processor count of a directory-based multiprocessor means that we cannot retain assumptions of unlimited buffering and must find new ways to avoid deadlock, Let’s start with the snooping protocols.

As we mentioned in Chapter 5, the challenge of implementing misses in a snooping coherence protocol without a bus lies in finding a way to make the multistep miss process appear atomic. Both an upgrade miss and a write miss require the same basic processing and generate the same implementation challenges; for simplicity, we focus on upgrade misses. Here are the steps in handling an upgrade miss:

1. Detect the miss and compose an invalidate message for transmission to other caches.
2. When access to the broadcast communication link is available, transmit the message.
3. When the invalidates have been processed, the processor updates the state of the cache block and then proceeds with the write that caused the upgrade miss.

There are two related difficulties that can arise. First, how will two processors, P1 and P2, that attempt to upgrade the same cache block at the same time resolve the race? Second, when at step 3, how does a processor know when all invalidates have been processed so that it can complete the step?

The solution to finding a winner in the race lies in the ordering imposed by the broadcast communication medium. The communication medium must broadcast any cache miss to all the nodes. If P1 and P2 attempt to broadcast at the same time, we must ensure that either P1's message will reach P2 first or P2's will reach P1 first. This property will be true if there is a single channel through which all ingoing and outgoing requests from a node must pass through and if the communication network does not accept a message unless it can guarantee delivery (i.e., it is effectively circuit switched, see Appendix F). If both P1 and P2 initiate their attempts to broadcast an invalidate simultaneously, then the network can accept only one of these operations and delay the other. This ordering ensures that either P1 or P2 will complete its communication in step 2 first. The network can explicitly signal when it accepts a message and can guarantee it will be the next transmission; alternatively, a processor can simply watch the network for its own request, knowing that once the request is seen, it will be fully transmitted to all processors before any subsequent messages.

Now, suppose P1 wins the race to transmit its invalidate; once it knows it has won the race, it can continue with step 3 and complete the miss handling. There is a potential problem, however, for P2. When P2 undertook step 1, it believed that the block was in the shared state, but for P1 to advance at step 3, it must know that P2 has processed the invalidate, which must change the state of the block at P2 to invalid! One simple solution is for P2 to notice that it has lost the race, by observing that P1's invalidate is broadcast before its own invalidate. P2 can then invalidate the block and generate a write miss to get the data. P1 will see its invalidate before P2's, so it will change the block to modified and update the data, which guarantees forward progress and avoids deadlock. When P1 sees the subsequent invalidate to a block in the Modified state (a possibility that cannot arise in our basic protocol discussed in Chapter 5), it knows that it was the winner of a race. It can simply ignore the invalidate, knowing that it will be followed by a write miss, or it can write the block back to memory and make its state invalid.

Another solution is to give precedence to incoming requests over outgoing requests, so that before P2 can transmit its invalidate it must handle any pending invalidates or write misses. If any of those misses are for blocks with the same address as a pending outgoing message, the processor must be prepared to restart the write operation, since the incoming request may cause the state of the block to change. Notice that P1 knows that the invalidates will be processed once it has successfully completed the broadcast, since precedence is given to invalidate messages over outgoing requests. (Because it does not employ broadcast, a processor using a directory protocol cannot know when an invalidate is received; instead, explicit acknowledgments are required, as we discuss in the next section. Indeed, as we will see, it cannot even know it has won the race to become the owner until its request is acknowledged.)

Reads will also require a multiple-step process, since we need to get the data back from memory or a remote cache (in a write-back cache system), but reads do not introduce fundamentally new problems beyond what exists for writes.

There are, however, a few additional tricky edge cases that must be handled correctly. For example, in a write-back cache, a processor can generate a read miss that requires a write-back, which it could delay, while giving the read miss priority. If a snoop request appears for the cache block that is to be written back, the processor must discover this and send the data back. Failure to do so can create a deadlock situation. A similar tricky situation exists when a processor generates a write miss, which will make a block exclusive, but, before the processor receives the data and can update the block, other processors generate read misses for that block. The read misses cannot be processed until the writing processor receives the data and updates the block.

One of the more difficult problems occurs in a write-back cache where the data for a read or write miss can come either from memory or from one of the processor caches, but the requesting processor will not know *a priori* where the data will come from. In most bus-based systems, a single global signal is used to indicate whether any processor has the exclusive (and hence the most up-to-date) copy; otherwise, the memory responds. These schemes can work with a pipelined interconnection by requiring that processors signal whether they have the exclusive copy within a fixed number of cycles after the miss is broadcast.

In a modern multiprocessor, however, it is essentially impossible to bound the amount of time required for a snoop request to be processed. Instead, a mechanism is required to determine whether the memory has an up-to-date copy. One solution is to add coherence bits to the memory, indicating whether the data are exclusive in a remote cache. This mechanism begins to move toward the directory approach, whose implementation challenges we consider next.

Implementing Cache Coherence in a DSM Multiprocessor

Implementing a directory-based cache coherence protocol requires overcoming all the problems related to nonatomic actions for a snooping protocol without the use of broadcast (see Chapter 5), which forced a serialization on competing writes and also ensured the serialization required for the memory consistency model. Avoiding the need to broadcast is a central goal for a directory-based system, so another method for ensuring serialization is necessary.

The serialization of requests for exclusive access to a memory block is easily enforced since those requests will be serialized when they reach the unique directory for the specified block. If the directory controller simply ensures that one request is completely serviced before the next is begun, writes will be serialized. Because the requesters cannot know ahead of time who will win the race and because the communication is not a broadcast, the directory must signal to the winner when it completes the processing of the winner's request. This is done by a message that supplies the data on a write miss or by an explicit acknowledgment message that grants ownership in response to an invalidation request.

What about the loser in this race? The simplest solution is for the system to send a *negative acknowledge*, or *NAK*, which requires that the requesting node regenerate its request. (This is the equivalent of a collision in the broadcast network in a snooping scheme, which requires that one of the transmitting nodes retry its communication.) We will see in the next section why the NAK approach, as opposed to buffering the request, is attractive.

Although the acknowledgment that a requesting node has ownership is completed when the write miss or ownership acknowledgment message is transmitted, we still do not know that the invalidates have been received and processed by the nodes that were in the sharing set. All memory consistency models eventually require (either before the next cache miss or at a synchronization point, for example) that a processor knows that all the invalidates for a write have been processed. In a snooping scheme, the nature of the broadcast network provides this assurance.

How can we know when the invalidates are complete in a directory scheme? The only way to know that the invalidates have been completed is to have the destination nodes of the invalidate messages (the members of the sharing set) explicitly acknowledge the invalidation messages sent from the directory. Who should they be acknowledged to? There are two possibilities. In the first the acknowledgments can be sent to the directory, which can count them, and when all acknowledgments have been received, confirm this with a single message to the original requester. Alternatively, when granting ownership, the directory can tell the register how many acknowledgments to expect. The destinations of the invalidate messages can then send an acknowledgment directly to the requester, whose identity is provided by the directory. Most existing implementations use the latter scheme, since it reduces the possibility of creating a bottleneck at a directory. Although the requirement for acknowledgments is an additional complexity in directory protocols, this requirement arises from the avoidance of a serialization mechanism, such as the snooping broadcast operation, which in itself is the limit to scalability.

Avoiding Deadlock from Limited Buffering

A new complication in the implementation is introduced by the potential scale of a directory-based multiprocessor. In Chapter 5, we assumed that the network could always accept a coherence message and that the request would be acted upon at some point. In a much larger multiprocessor, this assumption of unlimited buffering may be unreasonable. What happens when the network does not have unlimited buffering? The major implication of this limit is that a cache or directory controller may be unable to complete a message send. This could lead to deadlock.

The potential deadlock arises from three properties, which characterize many deadlock situations:

1. More than one resource is needed to complete a transaction: Message buffers are needed to generate requests, create replies and acknowledgments, and accept replies.
2. Resources are held until a nonatomic transaction completes: The buffer used to create the reply cannot be freed until the reply is accepted, for reasons we will see shortly.
3. There is no global partial order on the acquisition of resources: Nodes can generate requests and replies at will.

These characteristics lead to deadlock, and avoiding deadlock requires breaking one of these properties. Freeing up resources without completing a transaction is difficult, since the transaction must be completely backed out and cannot be left half-finished. Hence, our approach will be to try to resolve the need for multiple resources. We cannot simply eliminate this need, but we can try to ensure that the resources will always be available.

One way to ensure that a transaction can always complete is to guarantee that there are always buffers to accept messages. Although this is possible for a small multiprocessor with processors that block on a cache miss or have a small number of outstanding misses, it may not be very practical in a directory protocol, since a single write could generate many invalidate messages. In addition, features such as prefetch and multiple outstanding misses increase the amount of buffering required. There is an alternative strategy, which most systems use and which ensures that a transaction will not actually be initiated until we can guarantee that it has the resources to complete. The strategy has four parts:

1. A separate network (physical or virtual) is used for requests and replies, where a reply is any message that a controller waits for in transitioning between states. This ensures that new requests cannot block replies that will free up buffers.
2. Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.

3. Any controller can reject with a NAK any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.
4. Any request that receives a NAK in response is simply retried.

To see that there are no deadlocks with the four properties above, we must ensure that all replies can be accepted and that every request is eventually serviced. Since a cache controller or directory controller always allocates a buffer to handle the reply before issuing a request, it can always accept the reply when it returns. To see that every request is eventually serviced, we need only show that any request could be completed. Since every request starts with a read or write miss at a cache, it is sufficient to show that any read or write miss is eventually serviced. Since the write miss case includes the actions for a read miss as a subset, we focus on showing the write misses are serviced. The simplest situation is when the block is uncached; since that case is subsumed by the case when the block is shared, we focus on the shared and exclusive cases. Let's consider the case where the block is shared:

- The CPU attempts to do a write and generates a write miss that is sent to the directory. For simplicity, we can assume that the processor is stalled. Although it may issue further requests, it should not issue a request for the same cache block until the first one is completed. Requests for independent blocks can be handled separately.
- The write miss is sent to the directory controller for this memory block. Note that although one cache controller handles all the requests for a given cache block, regardless of its memory contents, the directory controller handles requests for different blocks as independent events (assuming sufficient buffering, which is allocated before the directory issues any further messages on behalf of the request). The only conflict at the directory controller is when two requests arrive for the same block. The controller must wait for the first operation to be completed. It can simply NAK the second request or buffer it, but it should not service the second request for a given memory block until the first is completed.
- Now consider what happens at the directory controller: Suppose the write miss is the next thing to arrive at the directory controller. The controller sends out the invalidates, which can always be accepted after a limited delay by the cache controller. Note that one possibility is that the cache controller has an outstanding miss for the same block. This is the dual case to the snooping scheme, and we must once again break the tie by forcing the cache controller to accept and act on the directory request. Depending on the exact timing, this cache controller will either get the cache line later from the directory or will receive a NAK and have to restart the process.

The case where the block is exclusive is somewhat trickier. Our analysis begins when the write miss arrives at the directory controller for processing. There are two cases to consider:

- The directory controller sends a fetch/invalidate message to the processor where it arrives to find the block in the exclusive state. The cache controller sends a data write-back to the home directory and makes its state invalid. This reply arrives at the home directory controller, which can always accept the reply, since it preallocated the buffer. The directory controller sends back the data to the requesting processor, which can always accept the reply; after the cache is updated, the requesting cache controller notifies the processor.
- The directory controller sends a fetch/invalidate message to the node indicated as owner. When the message arrives at the owner node, it finds that this cache controller has taken a read or write miss that caused the block to be replaced. In this case, the cache controller has already sent the block to the home directory with a data write-back and made the data unavailable. Since this is exactly the effect of the fetch/invalidate message, the protocol operates correctly in this case as well.

We have shown that our coherence mechanism operates correctly when the cache and directory controller can accept requests for operation on cache blocks for which they have no outstanding operations in progress, when replies are always accepted, and when requests can be NAKed and forced to retry. Like the case of the snooping protocol, the cache controller must be able to break ties, and it always does so by favoring the instructions from the directory. The ability to NAK requests is what allows an implementation with finite buffering to avoid deadlock.

Implementing the Directory Controller

To implement a cache coherence scheme, the cache controller must have the same abilities it needed in the snooping case, namely, the capability of handling requests for independent blocks while awaiting a response to a request from the local processor. The incoming requests are still processed in order, and each one is completed before beginning the next. Should a cache controller receive too many requests in a short period of time, it can NAK them, knowing that the directory will subsequently regenerate the request.

The directory must also be multithreaded and able to handle requests for multiple blocks independently. This situation is somewhat different than having the cache controller handle incoming requests for independent blocks, since the directory controller will need to begin processing one request while an earlier one is still underway. The directory controller cannot wait for one to complete before servicing the next request, since this could lead to deadlock. Instead, the directory controller must be *reentrant*; that is, it must be capable of suspending its execution while waiting for a reply and accepting another transaction. The only place this must occur is in response to read or write misses, while waiting for a response from the owner. This leads to three important observations:

1. The state of the controller need only be saved and restored while either a fetch operation from a remote location or a fetch/invalidate is outstanding.
2. The implementation can bound the number of outstanding transactions being handled in the directory by simply NAKing read or write miss requests that could cause the number of outstanding requests to be exceeded.
3. If instead of returning the data through the directory, the owner node forwards the data directly to the requester (as well as returning it to the directory), we can eliminate the need for the directory to handle more than one outstanding request. This motivation, in addition to the reduction of latency, is the reason for using the forwarding style of protocol. There are other complexities from forwarding protocols that arise when requests arrive closely spaced in time.

The major remaining implementation difficulty is to handle NAKs. One alternative is for each processor to keep track of its outstanding transactions so it knows, when the NAK is received, what the requested transaction was. The alternative is to bundle the original request into the NAK, so that the controller receiving the NAK can determine what the original request was. Because every request allocates a slot to receive a reply and a NAK is a reply, NAKs can always be received. In fact, the buffer holding the return slot for the request can also hold information about the request, allowing the processor to reissue the request if it is NAKed.

In practice, great care is required to implement these protocols correctly and to avoid deadlock. The key ideas we have seen in this section—dealing with nonatomicity and finite buffering—are critical to ensuring a correct implementation. Designers have found that both formal and informal verification techniques are helpful for ensuring that implementations are correct.

I.8 The Custom Cluster Approach: Blue Gene/L

Blue Gene/L (BG/L) is a scalable message-passing supercomputer whose design offers unprecedented computing density as measured by compute power per watt. By focusing on power efficiency, BG/L also achieves unmatched throughput per cubic foot. High computing density, combined with cost-effective nodes and extensive support for RAS, allows BG/L to efficiently scale to very large processor counts.

BG/L is a distributed-memory, message-passing computer but one that is quite different from the cluster-based, often throughput-oriented computers that rely on commodity technology in the processors, interconnect, and, sometimes, the packaging and system-level organization. BG/L uses a special customized processing node that contains two processors (derived from low-power, lower-clock-rate PowerPC 440 chips used in the embedded market), caches, and interconnect logic.

A complete computing node is formed by adding SDRAM chips, which are the only commodity semiconductor parts in the BG/L design.

BG/L consists of up to 64 K nodes organized into 32 racks each containing 1 K nodes in about 50 cubic feet. Each rack contains two double-sided boards with 512 nodes each. Due to the high density within a board and rack, 85% of the interconnect is within a single rack, greatly reducing the complexity and latency associated with connections between racks. Furthermore, the compact size of a rack, which is enabled by the low power and high density of each node, greatly improves efficiency, since the interconnection network for connections within a single rack are integrated into the single compute chip that comprises each node.

Appendix F discusses the main BL/G interconnect network, which is a three-dimensional torus. There are four other networks: Gigabit Ethernet, connected at designated I/O nodes; a JTAG network used for test; a barrier network; and a global collective network. The barrier network contains four independent channels and can be used for performing a global `or` or a global `and` across all the processors with latency of less than 1.5 microseconds. The global collective network connects all the processors in a tree and is used for global operations. It supports a variety of integer reductions directly, avoiding the need to involve the processor, and leading to times for large-scale reductions that are 10 to 100 times faster than in typical supercomputers. The collective network can also be used to broadcast a single value efficiently. Support for the collective network as well as the torus is included in the chip that forms of the heart of each processing node.

The Blue Gene/L Computing Node

Each BG/L node consists of a single processing chip and several SDRAM chips. The BG/L processing chip, shown in Figure I.18, contains the following:

1. Two PowerPC 440 CPUs, each a two-issue superscalar with a seven-stage pipeline and speculative out-order issue capability, clocked at a modest (and power-saving) 700 MHz. Each CPU has separate 32 KB I and D caches that are nonbblocking with up to four outstanding misses. Cache coherence must be enforced in software. Each CPU also contains a pair of floating-point coprocessors, each with its own FP register set and each capable of issuing a multiply-add each clock cycle, supporting a special SIMD instruction set capability that includes complex arithmetic using a pair of registers and 128-bit operands.
2. Separate fully associative L2 caches, each with 2 KB of data and a 128-byte block size, that act essentially like prefetch buffers. The L2 cache controllers recognize streamed data access and also handle prefetch from L3 or main memory. They have low latency (11 cycles) and provide high bandwidth (5 bytes per clock). The L2 prefetch buffer can supply 5.5 GB/sec to the L1 caches.
3. A 4 MB L3 cache implemented with embedded DRAM. Each L2 buffer is connected by a bus supplying 11 GB/sec of bandwidth from the L3 cache.

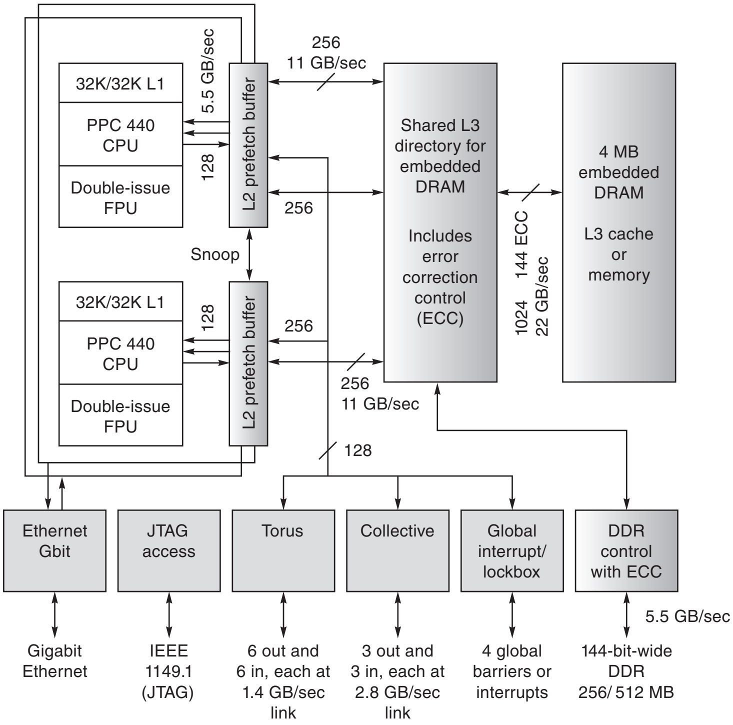

32K/32K L1
PPC 440 CPU
Double-issue FPU
5.5 GB/sec
128
L2 prefetch buffer
Snoop
32K/32K L1
PPC 440 CPU
Double-issue FPU
128
L2 prefetch buffer
256
11 GB/sec
256
256
256
11 GB/sec
128
Shared L3 directory for embedded DRAM
Includes error correction control (ECC)
1024 144 ECC
22 GB/sec
4 MB embedded DRAM
L3 cache or memory
Ethernet Gbit
JTAG access
Torus
Collective
Global interrupt/ lockbox
DDR control with ECC
5.5 GB/sec
Gigabit Ethernet
IEEE 1149.1 (JTAG)
6 out and 6 in, each at 1.4 GB/sec link
3 out and 3 in, each at 2.8 GB/sec link
4 global barriers or interrupts
144-bit-wide DDR 256/ 512 MB

Figure I.18 The BG/L processing node. The unfilled boxes are the PowerPC processors with added floating-point units. The solid gray boxes are network interfaces, and the shaded lighter gray boxes are part of the memory system, which is supplemented by DDR RAMS.

4. A memory bus supporting 256 to 512 MB of DDR DRAMS and providing 5.5 GB/sec of memory bandwidth to the L3 cache. This amount of memory might seem rather modest for each node, given that the node contains two processors, each with two FP units. Indeed Amdahl's rule of thumb (1 MB per 1 MIPS) and an assumption of 25% of peak performance would favor about 2.7 times the memory per node. For floating-point-intensive applications where the computational need usually grows faster than linear in the memory size, the upper limit of 512 MB/node is probably reasonable.
5. Support logic for the five interconnection networks.

By placing all the logic other than DRAMs into a single chip, BG/L achieves higher density, lower power, and lower cost, making it possible to pack the processing nodes extremely densely. The density in terms allows the interconnection networks to be low latency, high bandwidth, and quite cost effective. The combination yields a supercomputer that scales very cost-effectively, yielding an order-of-magnitude improvement

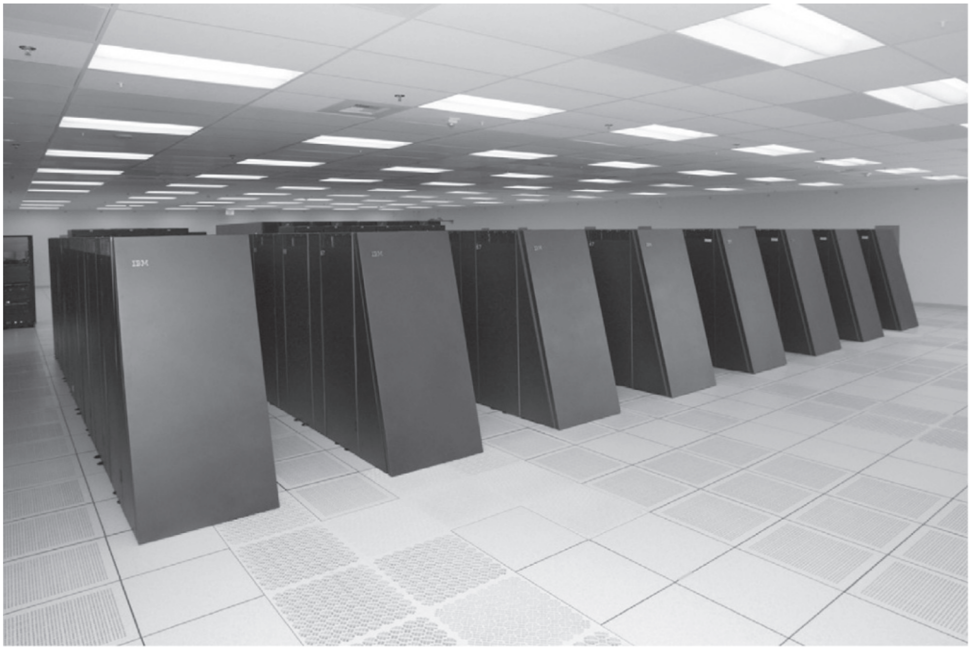


Figure I.19 The 64 K-processor Blue Gene/L system.

in GFLOPs/watt over other approaches as well as significant improvements in GFLOPS/$ for very large-scale multiprocessors.

For example, BG/L with 64 K nodes has a peak performance of 360 TF and uses about 1.4 megawatts. To achieve 360 TF peak using the Power5+, which is the most power-efficient, high-end FP processor, would require about 23,500 processors (the dual processor can execute up to 8 FLOPs/clock at 1.9 GHz). The power requirement for just the processors, without external cache, DRAM, or interconnect, would be about 2.9 megawatts, or about double the power of the entire BG/L system. Likewise, the smaller die size of the BG/L node and its need for DRAMs as the only external chip produce significant cost savings versus a node built using a high-end multiprocessor. Figure I.19 shows a photo of the 64K node BG/L. The total size occupied by this 128K-processor multiprocessor is comparable to that occupied by earlier multiprocessors with 16K processors.

I.9 Concluding Remarks

The landscape of large-scale multiprocessors has changed dramatically over the past five to ten years. While some form of clustering is now used for all the largest-scale multiprocessors, calling them all "clusters" ignores significant differences in architecture, implementation style, cost, and performance. Bell and Gray

Terminology	Characteristics	Examples
MPP	Originally referred to a class of architectures characterized by large numbers of small, typically custom processors and usually using an SIMD style architecture.	Connection Machines CM-2
SMP (symmetric multiprocessor)	Shared-memory multiprocessors with a symmetric relationship to memory; also called UMA (uniform memory access). Scalable versions of these architectures used multistage interconnection networks, typically configured with at most 64 to 128 processors.	SUN Sunfire, NEC Earth Simulator
DSM (distributed shared memory)	A class of architectures that support scalable shared memory in a distributed fashion. These architectures are available both with and without cache coherence and typically can support hundreds to thousands of processors.	SGI Origin and Altix, Cray T3E, Cray X1, IBM p5 590/5
Cluster	A class of multiprocessors using message passing. The individual nodes are either commodities or customized, likewise the interconnect.	See commodity and custom clusters
Commodity cluster	A class of clusters where the nodes are truly commodities, typically headless workstations, motherboards, or blade servers, connected with a SAN or LAN usually accessible via an I/O bus.	"Beowulf" and other "homemade" clusters
Custom cluster	A cluster architecture where the nodes and the interconnect are customized and more tightly integrated than in a commodity cluster. Also called distributed memory or message passing multiprocessors.	IBM Blue Gene, Cray XT3
Constellation	Large-scale multiprocessors that use clustering of smaller-scale multiprocessors, typically with a DSM or SMP architecture and 32 or more processors.	Larger SGI Origin/Altix, ASC Purple

Figure I.20 A classification of large-scale multiprocessors. The term *MPP,* which had the original meaning described above, has been used more recently, and less precisely, to refer to all large-scale multiprocessors. None of the commercial shipping multiprocessors is a true MPP in the original sense of the word, but such an approach may make sense in the future. Both the SMP and DSM class includes multiprocessors with vector support. The term *constellation* has been used in different ways; the above usage seems both intuitive and precise [Dongarra et al. 2005].

[2002] discussed this trend, arguing that clusters will dominate. While Dongarra et al. [2005] agreed that some form of clustering is almost inevitable in the largest multiprocessors, they developed a more nuanced classification that attempts to distinguish among a variety of different approaches.

In Figure I.20 we summarize the range of terminology that has been used for large-scale multiprocessors and focus on defining the terms from an architectural and implementation perspective. Figure I.21 shows the hierarchical relationship of these different architecture approaches. Although there has been some convergence in architectural approaches over the past 15 years, the TOP500 list, which reports the 500 fastest computers in the world as measured by the Linpack benchmark, includes commodity clusters, customized clusters, Symmetric Multiprocessors (SMPs), DSMs, and constellations, as well as processors that are both scalar and vector.

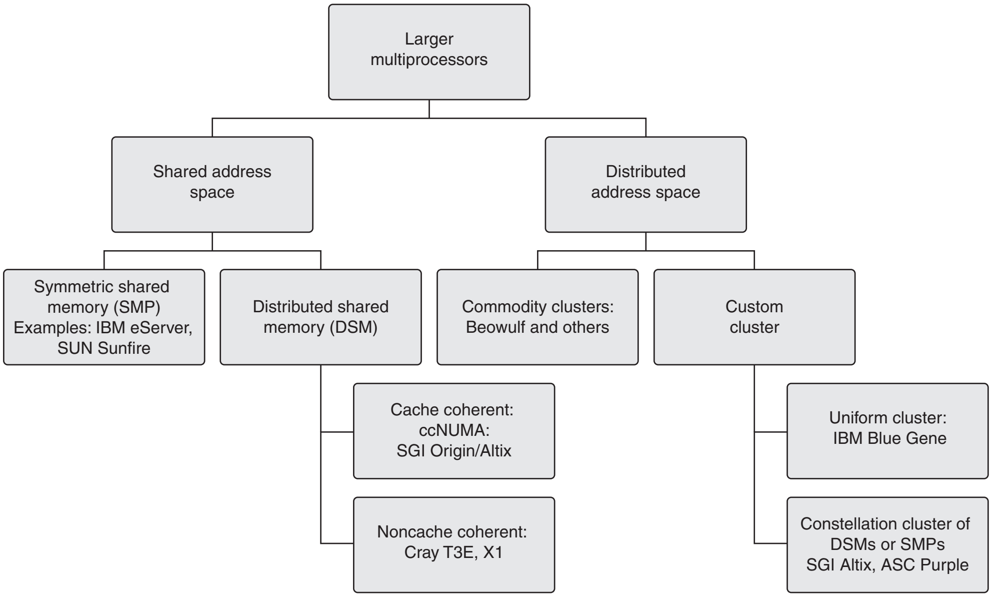

Larger multiprocessors
Shared address space
Distributed address space
Symmetric shared memory (SMP) Examples: IBM eServer, SUN Sunfire
Distributed shared memory (DSM)
Commodity clusters: Beowulf and others
Custom cluster
Cache coherent: ccNUMA: SGI Origin/Altix
Noncache coherent: Cray T3E, X1
Uniform cluster: IBM Blue Gene
Constellation cluster of DSMs or SMPs SGI Altix, ASC Purple

Figure I.21 The space of large-scale multiprocessors and the relation of different classes.

Nonetheless, there are some clearly emerging trends, which we can see by looking at the distribution of types of multiprocessors in the TOP500 list:

1. Clusters represent a majority of the systems. The lower development effort for clusters has clearly been a driving force in making them more popular. The high-end multiprocessor market has not grown sufficiently large to support full-scale, highly customized designs as the dominant choice.
2. The majority of the clusters are commodity clusters, often put together by users, rather than a system vendor designing a standard product.
3. Although commodity clusters dominate in their representation, the top 25 entries on the list are much more varied and include 9 custom clusters (primarily instances of Blue Gene or Cray XT3 systems), 2 constellations, 8 commodity clusters, 2 SMPs (one of which is the NEC Earth Simulator, which has nodes with vector processors), and 4 DSM multiprocessors.
4. Vector processors, which once dominated the list, have almost disappeared.
5. The IBM Blue Gene dominates the top 10 systems, showing the advantage of an approach the uses some commodity processor cores, but customizes many other functions and balances performance, power, and packaging density.
6. Architectural convergence has been driven more by market effects (lack of growth, limited suppliers, etc.) than by a clear-cut consensus on the best architectural approaches.

Software, both applications and programming languages and environments, remains the big challenge for parallel computing, just as it was 30 years ago, when multiprocessors such as the Illiac IV were being designed. The combination of ease of programming with high parallel performance remains elusive. Until better progress is made on this front, convergence toward a single programming model and underlying architectural approach (remembering that for uniprocessors we essentially have one programming model and one architectural approach!) will be slow or will be driven by factors other than proven architectural superiority.

J.1 Introduction J-2
J.2 Basic Techniques of Integer Arithmetic J-2
J.3 Floating Point J-13
J.4 Floating-Point Multiplication J-17
J.5 Floating-Point Addition J-21
J.6 Division and Remainder J-27
J.7 More on Floating-Point Arithmetic J-32
J.8 Speeding Up Integer Addition J-37
J.9 Speeding Up Integer Multiplication and Division J-44
J.10 Putting It All Together J-57
J.11 Fallacies and Pitfalls J-62
J.12 Historical Perspective and References J-63
Exercises J-67

J

Computer Arithmetic

by David Goldberg
Xerox Palo Alto Research Center

> The Fast drives out the Slow even if the Fast is wrong.
>
> W. Kahan

J.1 Introduction

Although computer arithmetic is sometimes viewed as a specialized part of CPU design, it is a very important part. This was brought home for Intel in 1994 when their Pentium chip was discovered to have a bug in the divide algorithm. This floating-point flaw resulted in a flurry of bad publicity for Intel and also cost them a lot of money. Intel took a $300 million write-off to cover the cost of replacing the buggy chips.

In this appendix, we will study some basic floating-point algorithms, including the division algorithm used on the Pentium. Although a tremendous variety of algorithms have been proposed for use in floating-point accelerators, actual implementations are usually based on refinements and variations of the few basic algorithms presented here. In addition to choosing algorithms for addition, subtraction, multiplication, and division, the computer architect must make other choices. What precisions should be implemented? How should exceptions be handled? This appendix will give you the background for making these and other decisions.

Our discussion of floating point will focus almost exclusively on the IEEE floating-point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating-point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms (but not necessarily sharing the hardware that implements integer instructions). Thus, after our discussion of floating point, we will take a more detailed look at integer algorithms.

Some good references on computer arithmetic, in order from least to most detailed, are Chapter 3 of Patterson and Hennessy [2009]; Chapter 7 of Hamacher, Vranesic, and Zaky [1984]; Gosling [1980]; and Scott [1985].

J.2 Basic Techniques of Integer Arithmetic

Readers who have studied computer arithmetic before will find most of this section to be review.

Ripple-Carry Addition

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are *half adders* and *full adders*. The half adder takes two bits a and b as input and produces a sum bit s and a carry bit c_{out} as output. Mathematically, $s = (a + b) \bmod 2$, and $c_{\text{out}} = \lfloor (a + b)/2 \rfloor$, where $\lfloor\ \rfloor$ is the floor function. As logic equations, $s = a\bar{b} + \bar{a}b$ and $c_{\text{out}} = ab$, where ab means $a \wedge b$ and $a + b$ means $a \vee b$. The half adder is also called a (2,2) adder, since it takes two inputs and produces two outputs. The full adder is a (3,2) adder and is defined by $s = (a + b + c) \bmod 2$, $c_{\text{out}} = \lfloor (a + b + c)/2 \rfloor$, or the logic equations

J.2.1 $$s = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

J.2.2 $$c_{\text{out}} = ab + ac + bc$$

The principal problem in constructing an adder for n-bit numbers out of smaller pieces is propagating the carries from one piece to the next. The most obvious way to solve this is with a *ripple-carry adder*, consisting of n full adders, as illustrated in Figure J.1. (In the figures in this appendix, the least-significant bit is always on the right.) The inputs to the adder are $a_{n-1}a_{n-2}\cdots a_0$ and $b_{n-1}b_{n-2}\cdots b_0$, where $a_{n-1}a_{n-2}\cdots a_0$ represents the number $a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_0$. The c_{i+1} output of the ith adder is fed into the c_{i+1} input of the next adder (the $(i+1)$-th adder) with the lower-order carry-in c_0 set to 0. Since the low-order carry-in is wired to 0, the low-order adder could be a half adder. Later, however, we will see that setting the low-order carry-in bit to 1 is useful for performing subtraction.

In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. However, determining the exact relationship between logic levels and timings is highly technology dependent. Therefore, when comparing adders we will simply compare the number of logic levels in each one. How many levels are there for a ripple-carry adder? It takes two levels to compute c_1 from a_0 and b_0. Then it takes two more levels to compute c_2 from c_1, a_1, b_1, and so on, up to c_n. So, there are a total of $2n$ levels. Typical values of n are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only n simple cells, connected in a simple, regular way.

Because the ripple-carry adder is relatively slow compared with the designs discussed in Section J.8, you might wonder why it is used at all. In technologies like CMOS, even though ripple adders take time $O(n)$, the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders.

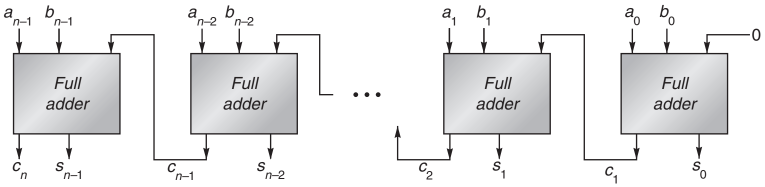


Figure J.1 Ripple-carry adder, consisting of *n* full adders. The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

Radix-2 Multiplication and Division

The simplest multiplier computes the product of two unsigned numbers, one bit at a time, as illustrated in Figure J.2(a). The numbers to be multiplied are $a_{n-1}a_{n-2}\cdots a_0$ and $b_{n-1}b_{n-2}\cdots b_0$, and they are placed in registers A and B, respectively. Register P is initially 0. Each multiply step has two parts:

Multiply Step

(*i*) If the least-significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2}\cdots b_0$, is added to P; otherwise, $00\cdots00$ is added to P. The sum is placed back into P.

(*ii*) Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.

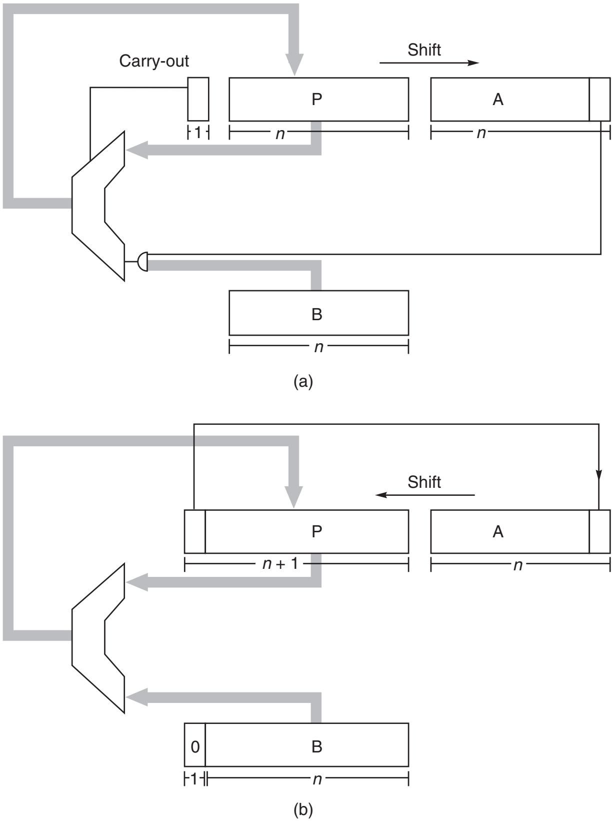


Figure J.2 Block diagram of (a) multiplier and (b) divider for *n*-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

After n steps, the product appears in registers P and A, with A holding the lower-order bits.

The simplest divider also operates on unsigned numbers and produces the quotient bits one at a time. A hardware divider is shown in Figure J.2(b). To compute a/b, put a in the A register, b in the B register, and 0 in the P register and then perform n divide steps. Each divide step consists of four parts:

Divide Step

(*i*) Shift the register pair (P,A) one bit left.

(*ii*) Subtract the content of register B (which is $b_{n-1}b_{n-2}\cdots b_0$) from register P, putting the result back into P.

(*iii*) If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.

(*iv*) If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this process n times, the A register will contain the quotient, and the P register will contain the remainder. This algorithm is the binary version of the paper-and-pencil method; a numerical example is illustrated in Figure J.3(a).

Notice that the two block diagrams in Figure J.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in Figure J.3(a) is called *restoring,* because if subtraction by b yields a negative result, the P register is restored by adding b back in. The restoring algorithm has a variant that skips the restoring step and instead works with the resulting negative numbers. Each step of this *nonrestoring* algorithm has three parts:

Nonrestoring

If P is negative,

Divide Step

(*i-a*) Shift the register pair (P,A) one bit left.

(*ii-a*) Add the contents of register B to P.

Else,

(*i-b*) Shift the register pair (P,A) one bit left.

(*ii-b*) Subtract the contents of register B from P.

(*iii*) If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this n times, the quotient is in A. If P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add b), and then it will be the remainder. A numerical example is given in Figure J.3(b). Since steps (*i-a*) and (*i-b*) are the same, you might be tempted to perform this common step first, and then test the sign of P. That doesn't work, since the sign bit can be lost when shifting.

P	A	
00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	step 1(i): shift.
−00011		step 1(ii): subtract.
−00010	110**0**	step 1(iii): result is negative, set quotient bit to 0.
00001	110**0**	step 1(iv): restore.
00011	10**0**	step 2(i): shift.
−00011		step 2(ii): subtract.
00000	10**01**	step 2(iii): result is nonnegative, set quotient bit to 1.
00001	0**01**	step 3(i): shift.
−00011		step 3(ii): subtract.
−00010	0**010**	step 3(iii): result is negative, set quotient bit to 0.
00001	0**010**	step 3(iv): restore.
00010	**010**	step 4(i): shift.
−00011		step 4(ii): subtract.
−00001	**0100**	step 4(iii): result is negative, set quotient bit to 0.
00010	**0100**	step 4(iv): restore. The quotient is 0100_2 and the remainder is 00010_2.

(a)

00000	1110	Divide 14 = 1110_2 by 3 = 11_2. B always contains 0011_2.
00001	110	step 1(i-b): shift.
+11101		step 1(ii-b): subtract b (add two's complement).
11110	110**0**	step 1(iii): P is negative, so set quotient bit to 0.
11101	10**0**	step 2(i-a): shift.
+00011		step 2(ii-a): add b.
00000	10**01**	step 2(iii): P is nonnegative, so set quotient bit to 1.
00001	0**01**	step 3(i-b): shift.
+11101		step 3(ii-b): subtract b.
11110	0**010**	step 3(iii): P is negative, so set quotient bit to 0.
11100	**010**	step 4(i-a): shift.
+00011		step 4(ii-a): add b.
11111	**0100**	step 4(iii): P is negative, so set quotient bit to 0.
+00011		Remainder is negative, so do final restore step.
00010		The quotient is 0100_2 and the remainder is 00010_2.

(b)

Figure J.3 Numerical example of (a) restoring division and (b) nonrestoring division.

The explanation for why the nonrestoring algorithm works is this. Let r_k be the contents of the (P,A) register pair at step k, ignoring the quotient bits (which are simply sharing the unused bits of register A). In Figure J.3(a), initially A contains 14, so $r_0 = 14$. At the end of the first step, $r_1 = 28$, and so on. In the restoring algorithm, part (i) computes $2r_k$ and then part (ii) $2r_k - 2^n b$ ($2^n b$ since b is subtracted from the left half). If $2r_k - 2^n b \geq 0$, both algorithms end the step with identical values in (P,A). If $2r_k - 2^n b < 0$, then the restoring algorithm restores this to $2r_k$, and the next step begins by computing $r_{\text{res}} = 2(2r_k) - 2^n b$. In the non-restoring algorithm, $2r_k - 2^n b$ is kept as a negative number, and in the next step $r_{\text{nonres}} = 2(2r_k - 2^n b) + 2^n b = 4r_k - 2^n b = r_{\text{res}}$. Thus (P,A) has the same bits in both algorithms.

If a and b are unsigned n-bit numbers, hence in the range $0 \leq a,b \leq 2^n - 1$, then the multiplier in Figure J.2 will work if register P is n bits long. However, for division, P must be extended to $n+1$ bits in order to detect the sign of P. Thus the adder must also have $n+1$ bits.

Why would anyone implement restoring division, which uses the same hardware as nonrestoring division (the control is slightly different) but involves an extra addition? In fact, the usual implementation for restoring division doesn't actually perform an add in step (*iv*). Rather, the sign resulting from the subtraction is tested at the output of the adder, and only if the sum is nonnegative is it loaded back into the P register.

As a final point, before beginning to divide, the hardware must check to see whether the divisor is 0.

Signed Numbers

There are four methods commonly used to represent signed n-bit numbers: *sign magnitude*, *two's complement*, *one's complement*, and *biased*. In the sign magnitude system, the high-order bit is the sign bit, and the low-order $n-1$ bits are the magnitude of the number. In the two's complement system, a number and its negative add up to 2^n. In one's complement, the negative of a number is obtained by complementing each bit (or, alternatively, the number and its negative add up to $2^n - 1$). In each of these three systems, nonnegative numbers are represented in the usual way. In a biased system, nonnegative numbers do not have their usual representation. Instead, all numbers are represented by first adding them to the bias and then encoding this sum as an ordinary unsigned number. Thus, a negative number k can be encoded as long as $k + \text{bias} \geq 0$. A typical value for the bias is 2^{n-1}.

Example Using 4-bit numbers ($n = 4$), if $k = 3$ (or in binary, $k = 0011_2$), how is $-k$ expressed in each of these formats?

Answer In signed magnitude, the leftmost bit in $k = \mathbf{0}011_2$ is the sign bit, so flip it to 1: $-k$ is represented by $\mathbf{1}011_2$. In two's complement, $k + 1101_2 = 2^n = 16$. So $-k$ is represented by 1101_2. In one's complement, the bits of $k = 0011_2$ are flipped, so $-k$ is represented by 1100_2. For a biased system, assuming a bias of $2^{n-1} = 8$, k is represented by $k + \text{bias} = 1011_2$, and $-k$ by $-k + \text{bias} = 0101_2$.

The most widely used system for representing integers, two's complement, is the system we will use here. One reason for the popularity of two's complement is that it makes signed addition easy: Simply discard the carry-out from the highorder bit. To add $5+-2$, for example, add 0101_2 and 1110_2 to obtain 0011_2, resulting in the correct value of 3. A useful formula for the value of a two's complement number $a_{n-1}a_{n-2}\cdots a_1a_0$ is

J.2.3 $$-a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_1 2^1+a_0$$

As an illustration of this formula, the value of 1101_2 as a 4-bit two's complement number is $-1\cdot 2^3+1\cdot 2^2+0\cdot 2^1+1\cdot 2^0=-8+4+1=-3$, confirming the result of the example above.

Overflow occurs when the result of the operation does not fit in the representation being used. For example, if unsigned numbers are being represented using 4 bits, then $6=0110_2$ and $11=1011_2$. Their sum (17) overflows because its binary equivalent (10001_2) doesn't fit into 4 bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is a carry-out of the most-significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into the high-order bit is different from the (to be discarded) carry-out of the high-order bit. In the example of $5+-2$ above, a 1 is carried both into and out of the leftmost bit, avoiding overflow.

Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate 0011_2, complement it to get 1100_2 and then add 1 to get 1101_2. Thus, to implement $a-b$ using an adder, simply feed a and $\overline{b}$ (where $\overline{b}$ is the number obtained by complementing each bit of b) into the adder and set the low-order, carry-in bit to 1. This explains why the rightmost adder in Figure J.1 is a full adder.

Multiplying two's complement numbers is not quite as simple as adding them. The obvious approach is to convert both operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result. Although this is conceptually simple, it requires extra time and hardware. Here is a better approach: Suppose that we are multiplying a times b using the hardware shown in Figure J.2(a). Register A is loaded with the number *a;* B is loaded with b. Since the content of register B is always b, we will use B and b interchangeably. If B is potentially negative but A is nonnegative, the only change needed to convert the unsigned multiplication algorithm into a two's complement one is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P (rather than the carry-out from the addition). Note that our n-bit-wide adder will now be adding n-bit two's complement numbers between -2^{n-1} and $2^{n-1}-1$.

Next, suppose a is negative. The method for handling this case is called *Booth recoding*. Booth recoding is a very basic technique in computer arithmetic and will play a key role in Section J.9. The algorithm on page J-4 computes $a\times b$ by examining the bits of a from least significant to most significant. For example, if $a=7=0111_2$, then step (*i*) will successively add B, add B, add B, and add 0. Booth recoding "recodes" the number 7 as $8-1=1000_2-0001_2=100\overline{1}$, where $\overline{1}$

represents -1. This gives an alternative way to compute $a \times b$, namely, successively subtract B, add 0, add 0, and add B. This is more complicated than the unsigned algorithm on page J-4, since it uses both addition and subtraction. The advantage shows up for negative values of a. With the proper recoding, we can treat a as though it were unsigned. For example, take $a = -4 = 1100_2$. Think of 1100_2 as the unsigned number 12, and recode it as $12 = 16 - 4 = 10000_2 - 0100_2 = 10\overline{1}00$. If the multiplication algorithm is only iterated n times ($n = 4$ in this case), the high-order digit is ignored, and we end up subtracting $0100_2 = 4$ times the multiplier—exactly the right answer. This suggests that multiplying using a recoded form of a will work equally well for both positive and negative numbers. And, indeed, to deal with negative values of a, all that is required is to sometimes subtract b from P, instead of adding either b or 0 to P. Here are the precise rules: If the initial content of A is $a_{n-1} \cdots a_0$, then at the ith multiply step the low-order bit of register A is a_i, and step (*i*) in the multiplication algorithm becomes:

I. If $a_i = 0$ and $a_{i-1} = 0$, then add 0 to P.
II. If $a_i = 0$ and $a_{i-1} = 1$, then add B to P.
III. If $a_i = 1$ and $a_{i-1} = 0$, then subtract B from P.
IV. If $a_i = 1$ and $a_{i-1} = 1$, then add 0 to P.

For the first step, when $i = 0$, take a_{i-1} to be 0.

Example When multiplying -6 times -5, what is the sequence of values in the (P,A) register pair?

Answer See Figure J.4.

P	A	
0000	1010	Put $-6 = 1010_2$ into A, $-5 = 1011_2$ into B.
0000	1010	step 1(i): $a_0 = a_{-1} = 0$, so from rule I add 0.
0000	0101	step 1(ii): shift.
+0101		step 2(i): $a_1 = 1$, $a_0 = 0$.Rule III says subtract b (or add $-b = -1011_2 = 0101_2$).
0101	0101	
0010	1010	step 2(ii): shift.
+ 1011		step 3(i): $a_2 = 0$, $a_1 = 1$. Rule II says add b (1011).
1101	1010	
1110	1101	step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)
+ 0101		step 4(i): $a_3 = 1$, $a_2 = 0$. Rule III says subtract b.
0011	1101	
0001	1110	step 4(ii): shift. Final result is $00011110_2 = 30$.

Figure J.4 Numerical example of Booth recoding. Multiplication of $a = -6$ by $b = -5$ to get 30.

The four prior cases can be restated as saying that in the *i*th step you should add $(a_{i-1} - a_i)$B to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b\left(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \ldots + a_1 2 + a_0\right) + ba_{-1}$$

Using Equation J.2.3 (page J-8) together with $a_{-1} = 0$, the right-hand side is seen to be the value of $b \times a$ as a two's complement number.

The simplest way to implement the rules for Booth recoding is to extend the A register one bit to the right so that this new bit will contain a_{i-1}. Unlike the naive method of inverting any negative operands, this technique doesn't require extra steps or any special casing for negative operands. It has only slightly more control logic. If the multiplier is being shared with a divider, there will already be the capability for subtracting b, rather than adding it. To summarize, a simple method for handling two's complement multiplication is to pay attention to the sign of P when shifting it right, and to save the most recently shifted-out bit of A to use in deciding whether to add or subtract b from P.

Booth recoding is usually the best method for designing multiplication hardware that operates on signed numbers. For hardware that doesn't directly implement it, however, performing Booth recoding in software or microcode is usually too slow because of the conditional tests and branches. If the hardware supports arithmetic shifts (so that negative b is handled correctly), then the following method can be used. Treat the multiplier a as if it were an unsigned number, and perform the first $n-1$ multiply steps using the algorithm on page J-4. If $a < 0$ (in which case there will be a 1 in the low-order bit of the A register at this point), then subtract b from P; otherwise ($a \geq 0$), neither add nor subtract. In either case, do a final shift (for a total of n shifts). This works because it amounts to multiplying b by $-a_{n-1}2^{n-1} + \cdots + a_1 2 + a_0$, which is the value of $a_{n-1} \cdots a_0$ as a two's complement number by Equation J.2.3. If the hardware doesn't support arithmetic shift, then converting the operands to be nonnegative is probably the best approach.

Two final remarks: A good way to test a signed-multiply routine is to try $-2^{n-1} \times -2^{n-1}$, since this is the only case that produces a $2n-1$ bit result. Unlike multiplication, division is usually performed in hardware by converting the operands to be nonnegative and then doing an unsigned divide. Because division is substantially slower (and less frequent) than multiplication, the extra time used to manipulate the signs has less impact than it does on multiplication.

Systems Issues

When designing an instruction set, a number of issues related to integer arithmetic need to be resolved. Several of them are discussed here.

First, what should be done about integer overflow? This situation is complicated by the fact that detecting overflow differs depending on whether the operands are signed or unsigned integers. Consider signed arithmetic first. There are three approaches: Set a bit on overflow, trap on overflow, or do nothing on overflow. In the last case, software has to check whether or not an overflow occurred. The most convenient solution for the programmer is to have an enable bit. If this bit is turned on, then overflow causes a trap. If it is turned off, then overflow sets a bit (or, alternatively, have two different add instructions). The advantage of this approach is that both trapping and nontrapping operations require only one instruction. Furthermore, as we will see in Section J.7, this is analogous to how the IEEE floating-point standard handles floating-point overflow. Figure J.5 shows how some common machines treat overflow.

What about unsigned addition? Notice that none of the architectures in Figure J.5 traps on unsigned overflow. The reason for this is that the primary use of unsigned arithmetic is in manipulating addresses. It is convenient to be able to subtract from an unsigned address by adding. For example, when $n=4$, we can subtract 2 from the unsigned address $10=1010_2$ by adding $14=1110_2$. This generates an overflow, but we would not want a trap to be generated.

A second issue concerns multiplication. Should the result of multiplying two n-bit numbers be a $2n$-bit result, or should multiplication just return the low-order n bits, signaling overflow if the result doesn't fit in n bits? An argument in favor of an n-bit result is that in virtually all high-level languages, multiplication is an operation in which arguments are integer variables and the result is an integer variable of the same type. Therefore, compilers won't generate code that utilizes a double-precision result. An argument in favor of a $2n$-bit result is that it can be used by an assembly language routine to substantially speed up multiplication of multiple-precision integers (by about a factor of 3).

A third issue concerns machines that want to execute one instruction every cycle. It is rarely practical to perform a multiplication or division in the same amount of time that an addition or register-register move takes. There are three possible approaches to this problem. The first is to have a single-cycle *multiply-step* instruction. This might do one step of the Booth algorithm. The second approach is to do integer multiplication in the floating-point unit and have it be part of the floating-point instruction set.

Machine	Trap on signed overflow?	Trap on unsigned overflow?	Set bit on signed overflow?	Set bit on unsigned overflow?
VAX	If enable is on	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
IBM 370	If enable is on	No	Yes. `Add` sets cond code.	Yes. Logical `add` sets cond code.
Intel 8086	No	No	Yes. `Add` sets V bit.	Yes. `Add` sets C bit.
MIPS R3000	Two `add` instructions; one always traps, the other never does.	No	No. Software must deduce it from sign of operands and result.	
SPARC	No	No	`Addcc` sets V bit. `Add` does not.	`Addcc` sets C bit. `Add` does not.

Figure J.5 Summary of how various machines handle integer overflow. Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

(This is what DLX does.) The third approach is to have an autonomous unit in the CPU do the multiplication. In this case, the result either can be guaranteed to be delivered in a fixed number of cycles—and the compiler charged with waiting the proper amount of time—or there can be an interlock. The same comments apply to division as well. As examples, the original SPARC had a multiply-step instruction but no divide-step instruction, while the MIPS R3000 has an autonomous unit that does multiplication and division (newer versions of the SPARC architecture added an integer multiply instruction). The designers of the HP Precision Architecture did an especially thorough job of analyzing the frequency of the operands for multiplication and division, and they based their multiply and divide steps accordingly. (See Magenheimer et al. [1988] for details.)

The final issue involves the computation of integer division and remainder for negative numbers. For example, what is -5 DIV 3 and -5 MOD 3? When computing x DIV y and x MOD y, negative values of x occur frequently enough to be worth some careful consideration. (On the other hand, negative values of y are quite rare.) If there are built-in hardware instructions for these operations, they should correspond to what high-level languages specify. Unfortunately, there is no agreement among existing programming languages. See Figure J.6.

One definition for these expressions stands out as clearly superior, namely, $x \text{ DIV } y = \lfloor x/y \rfloor$, so that $5 \text{ DIV } 3 = 1$ and $-5 \text{ DIV } 3 = -2$. And MOD should satisfy $x = (x \text{ DIV } y) \times y + x \text{ MOD } y$, so that $x \text{ MOD } y \geq 0$. Thus, $5 \text{ MOD } 3 = 2$, and $-5 \text{ MOD } 3 = 1$. Some of the many advantages of this definition are as follows:

1. A calculation to compute an index into a hash table of size N can use MOD N and be guaranteed to produce a valid index in the range from 0 to $N-1$.
2. In graphics, when converting from one coordinate system to another, there is no "glitch" near 0. For example, to convert from a value x expressed in a system that uses 100 dots per inch to a value y on a bitmapped display with 70 dots per inch, the formula $y = (70 \times x)$ DIV 100 maps one or two x coordinates into each y coordinate. But if DIV were defined as in Pascal to be x/y rounded to 0, then 0 would have three different points (-1, 0, 1) mapped into it.
3. x MOD 2^k is the same as performing a bitwise AND with a mask of k bits, and x DIV 2^k is the same as doing a k-bit arithmetic right shift.

Language	**Division**	**Remainder**
FORTRAN	$-5/3 = -1$	MOD$(-5, 3) = -2$
Pascal	-5 DIV $3 = -1$	-5 MOD $3 = 1$
Ada	$-5/3 = -1$	-5 MOD $3 = 1$ -5 REM $3 = -2$
C	$-5/3$ undefined	-5% 3 undefined
Modula-3	-5 DIV $3 = -2$	-5 MOD $3 = 1$

Figure J.6 Examples of integer division and integer remainder in various programming languages.

Finally, a potential pitfall worth mentioning concerns multiple-precision addition. Many instruction sets offer a variant of the `add` instruction that adds three operands: two n-bit numbers together with a third single-bit number. This third number is the carry from the previous addition. Since the multiple-precision number will typically be stored in an array, it is important to be able to increment the array pointer without destroying the carry bit.

J.3 Floating Point

Many applications require numbers that aren't integers. There are a number of ways that nonintegers can be represented. One is to use *fixed point*; that is, use integer arithmetic and simply imagine the binary point somewhere other than just to the right of the least-significant digit. Adding two such numbers can be done with an integer `add`, whereas multiplication requires some extra shifting. Other representations that have been proposed involve storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers (a,b) to represent the fraction a/b. However, only one noninteger representation has gained widespread use, and that is *floating point.* In this system, a computer word is divided into two parts, an exponent and a significand. As an example, an exponent of -3 and a significand of 1.5 might represent the number $1.5 \times 2^{-3} = 0.1875$. The advantages of standardizing a particular representation are obvious. Numerical analysts can build up high-quality software libraries, computer designers can develop techniques for implementing high-performance hardware, and hardware vendors can build standard accelerators. Given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use.

The semantics of floating-point instructions are not as clear-cut as the semantics of the rest of the instruction set, and in the past the behavior of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and overflow. Computer architecture books used to dispense advice on how to deal with all these details, but fortunately this is no longer necessary. That's because the computer industry is rapidly converging on the format specified by IEEE standard 754-1985 (also an international standard, IEC 559). The advantages of using a standard variant of floating point are similar to those for using floating point over other noninteger representations.

IEEE arithmetic differs from many previous arithmetics in the following major ways:

1. When rounding a "halfway" result to the nearest floating-point number, it picks the one that is even.
2. It includes the *special values* NaN, ∞, and $-\infty$.

3. It uses *denormal* numbers to represent the result of computations whose value is less than $1.0 \times 2^{E_{\min}}$.
4. It rounds to nearest by default, but it also has three other rounding modes.
5. It has sophisticated facilities for handling exceptions.

To elaborate on (1), note that when operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number. For example, in a floating-point system using base 10 and two significant digits, $6.1 \times 0.5 = 3.05$. This needs to be rounded to two digits. Should it be rounded to 3.0 or 3.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 3.05 rounds to 3.0, not 3.1. The standard actually has four *rounding modes*. The default is *round to nearest*, which rounds ties to an even number as just explained. The other modes are round toward 0, round toward $+\infty$, and round toward $-\infty$.

We will elaborate on the other differences in following sections. For further reading, see IEEE [1985], Cody et al. [1984], and Goldberg [1991].

Special Values and Denormals

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a *NaN* (*N*ot *a N*umber), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful, consider the code for a zero finder that takes a function F as an argument and evaluates F at various points to determine a zero for it. If the zero finder accidentally probes outside the valid values for F, then F may well cause an exception. Writing a zero finder that deals with this case is highly language and operating-system dependent, because it relies on how the operating system reacts to exceptions and how this reaction is mapped back into the programming language. In IEEE arithmetic it is easy to write a zero finder that handles this situation and runs on many different systems. After each evaluation of F, it simply checks to see whether F has returned a NaN; if so, it knows it has probed outside the domain of F.

In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.g., $3 + \text{NaN} = \text{NaN}$). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks. For example, suppose that arccos is computed in terms of arctan, using the formula $\arccos x = 2 \arctan\left(\sqrt{(1-x)/(1+x)}\right)$. If arctan handles an argument of NaN properly, arccos will automatically do so, too. That's because if x is a NaN, $1 + x$, $1 - x$, $(1+x)/(1-x)$, and $\sqrt{(1-x)/(1+x)}$ will also be NaNs. No checking for NaNs is required.

While the result of $\sqrt{-1}$ is a NaN, the result of 1/0 is not a NaN, but $+\infty$, which is another special value. The standard defines arithmetic on infinities (there are both $+\infty$ and $-\infty$) using rules such as $1/\infty = 0$. The formula $\arccos x = 2\arctan\left(\sqrt{(1-x)/(1+x)}\right)$ illustrates how infinity arithmetic can be used. Since arctan x asymptotically approaches $\pi/2$ as x approaches ∞, it is natural to define $\arctan(\infty) = \pi/2$, in which case $\arccos(-1)$ will automatically be computed correctly as $2\arctan(\infty) = \pi$.

The final kind of special values in the standard are *denormal* numbers. In many floating-point systems, if $E_{\min}$ is the smallest exponent, a number less than $1.0 \times 2^{E_{\min}}$ cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand, numbers less than $1.0 \times 2^{E_{\min}}$ are represented using significands less than 1. This is called *gradual underflow*. Thus, as numbers decrease in magnitude below $2^{E_{\min}}$, they gradually lose their significance and are only represented by 0 when all their significance has been shifted out. For example, in base 10 with four significant figures, let $x = 1.234 \times 10^{E_{\min}}$. Then, $x/10$ will be rounded to $0.123 \times 10^{E_{\min}}$, having lost a digit of precision. Similarly $x/100$ rounds to $0.012 \times 10^{E_{\min}}$, and $x/1000$ to $0.001 \times 10^{E_{\min}}$, while $x/10000$ is finally small enough to be rounded to 0. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as $x = y \Leftrightarrow x - y = 0$. For example, in a flush-to-zero system (again in base 10 with four significant digits), if $x = 1.256 \times 10^{E_{\min}}$ and $y = 1.234 \times 10^{E_{\min}}$, then $x - y = 0.022 \times 10^{E_{\min}}$, which flushes to zero. So even though $x \neq y$, the computed value of $x - y = 0$. This never happens with gradual underflow. In this example, $x - y = 0.022 \times 10^{E_{\min}}$ is a denormal number, and so the computation of $x - y$ is exact.

Representation of Floating-Point Numbers

Let us consider how to represent single-precision numbers in IEEE arithmetic. Single-precision numbers are stored in 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method (see the subsection "Signed Numbers," page J-7) with a bias of 127. The term *biased exponent* refers to the unsigned number contained in bits 1 through 8, and *unbiased exponent* (or just exponent) means the actual power to which 2 is to be raised. The fraction represents a number less than 1, but the *significand* of the floating-point number is 1 plus the fraction part. In other words, if e is the biased exponent (value of the exponent field) and f is the value of the fraction field, the number being represented is $1.f \times 2^{e-127}$.

Example What single-precision number does the following 32-bit word represent?

1 10000001 01000000000000000000000

Answer Considered as an unsigned number, the exponent field is 129, making the value of the exponent $129 - 127 = 2$. The fraction part is $.01_2 = .25$, making the significand 1.25. Thus, this bit pattern represents the number $-1.25 \times 2^2 = -5$.

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand $1.f$ does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part is usually *unpacked*, which is to say the implicit 1 is made explicit.

Figure J.7 summarizes the parameters for single (and other) precisions. It shows the exponents for single precision to range from -126 to 127; accordingly, the biased exponents range from 1 to 254. The biased exponents of 0 and 255 are used to represent special values. This is summarized in Figure J.8. When the biased exponent is 255, a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are 0, then the number represented is 0. Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to 1. Thus, a special convention such as this is required to represent 0. Denormalized numbers are implemented by having a word with a zero exponent field represent the number $0.f \times 2^{E_{\min}}$.

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that 0 is represented by a word of all 0s. The downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

	Single	**Single extended**	**Double**	**Double extended**
p (bits of precision)	24	≥ 32	53	≥ 64
$E_{\max}$	127	≥ 1023	1023	≥ 16383
$E_{\min}$	-126	≤ -1022	-1022	≤ -16382
Exponent bias	127		1023	

Figure J.7 Format parameters for the IEEE 754 floating-point standard. The first row gives the number of bits in the significand. The blanks are unspecified parameters.

Exponent	**Fraction**	**Represents**
$e = E_{\min} - 1$	$f = 0$	± 0
$e = E_{\min} - 1$	$f \neq 0$	$0.f \times 2^{E_{\min}}$
$E_{\min} \leq e \leq E_{\max}$	—	$1.f \times 2^{e}$
$e = E_{\max} + 1$	$f = 0$	$\pm\infty$
$e = E_{\max} + 1$	$f \neq 0$	NaN

Figure J.8 Representation of special values. When the exponent of a number falls outside the range $E_{\min} \leq e \leq E_{\max}$, then that number has a special interpretation as indicated in the table.

J.4 Floating-Point Multiplication

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number x is represented as a significand and an exponent, $x = s \times 2^e$. The formula

$$(s_1 \times 2^{e1}) \bullet (s_2 \times 2^{e2}) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating-point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned p-bit numbers (e.g., $p = 24$ for single precision), then the product can have as many as $2p$ bits and must be rounded to a p-bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents.

Example How does the multiplication of the single-precision numbers

$$\begin{array}{lcl} 1\ \ 1000001\ \ 0000\ldots & = & -1 \times 2^3 \\ 0\ \ 1000001\ \ 1000\ldots & = & 1 \times 2^4 \end{array}$$

proceed in binary?

Answer When unpacked, the significands are both 1.0, their product is 1.0, and so the result is of the form:

$$1\ \ ????????\ \ 000\ldots$$

To compute the exponent, use the formula:

$$\text{biased exp}(e_1 + e_2) = \text{biased exp}(e_1) + \text{biased exp}(e_2) - \text{bias}$$

From Figure J.7, the bias is $127 = 01111111_2$, so in two's complement -127 is 10000001_2. Thus, the biased exponent of the product is

$$\begin{array}{r} 10000010 \\ 10000011 \\ +\ 10000001 \\ \hline 10000110 \end{array}$$

Since this is 134 decimal, it represents an exponent of $134 - \text{bias} = 134 - 127$, as expected.

The interesting part of floating-point multiplication is rounding. Some of the different cases that can occur are illustrated in Figure J.9. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2.

$$\begin{array}{lrl}
\text{(a)} & 1.23 & \\
 & \underline{\times\ 6.78} & r=9>5 \text{ so round up} \\
 & \mathbf{8.33}94 & \text{rounds to } 8.34 \\
 & \uparrow\ \ & \\
\text{(b)} & 2.83 & \\
 & \underline{\times\ 4.47} & r=5 \text{ and a following digit} \neq 0 \text{ so round up} \\
 & \mathbf{12.6}501 & \text{rounds to } 1.27 \times 10^1 \\
 & \uparrow\ \ \ & \\
\text{(c)} & 1.28 & \\
 & \underline{\times\ 7.81} & r=6>5 \text{ so round up} \\
 & 0\mathbf{9.99}68 & \text{rounds to } 1.00 \times 10^1 \\
 & \uparrow\ \ &
\end{array}$$

Figure J.9 Examples of rounding a multiplication. Using base 10 and $p=3$, parts (a) and (b) illustrate that the result of a multiplication can have either $2p-1$ or $2p$ digits; hence, the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

In the figure, $p=3$, so the final result must be rounded to three significant digits. The three most-significant digits are in boldface. The fourth most-significant digit (marked with an arrow) is the *round* digit, denoted by r.

If the round digit is less than 5, then the bold digits represent the rounded result. If the round digit is greater than 5 (as in part (a)), then 1 must be added to the least-significant bold digit. If the round digit is exactly 5 (as in part (b)), then additional digits must be examined to decide between truncation or incrementing by 1. It is only necessary to know if any digits past 5 are nonzero. In the algorithm below, this will be recorded in a *sticky bit*. Comparing parts (a) and (b) in the figure shows that there are two possible positions for the round digit (relative to the least-significant digit of the product). Case (c) illustrates that, when adding 1 to the least-significant bold digit, there may be a carry-out. When this happens, the final significand must be 10.0.

There is a straightforward method of handling rounding using the multiplier of Figure J.2 (page J-4) together with an extra sticky bit. If p is the number of bits in the significand, then the A, B, and P registers should be p bits wide. Multiply the two significands to obtain a $2p$-bit product in the (P,A) registers (see Figure J.10). During the multiplication, the first $p-2$ times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let s represent the sticky bit, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A. There are two cases:

1. The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.
2. The high-order bit of P is 1. Set $s := s \vee r$ and $r := g$, and add 1 to the exponent.

Now if $r=0$, P is the correctly rounded product. If $r=1$ and $s=1$, then P + 1 is the product (where by P + 1 we mean adding 1 to the least-significant bit of P).

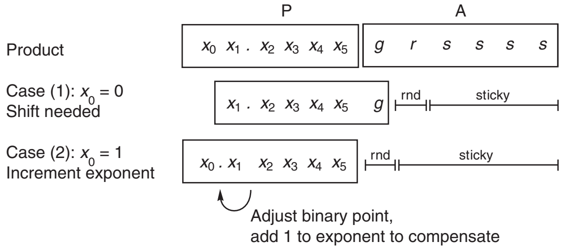

P
A
Product
x_0 x_1 . x_2 x_3 x_4 x_5
g r s s s s
Case (1): $x_0 = 0$
Shift needed
x_1 . x_2 x_3 x_4 x_5 g
rnd
sticky
Case (2): $x_0 = 1$
Increment exponent
x_0 . x_1 x_2 x_3 x_4 x_5
rnd
sticky
Adjust binary point,
add 1 to exponent to compensate

Figure J.10 The two cases of the floating-point multiply algorithm. The top line shows the contents of the P and A registers after multiplying the significands, with $p = 6$. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked *s*.

If $r = 1$ and $s = 0$, we are in a halfway case and round up according to the least-significant bit of P. As an example, apply the decimal version of these rules to Figure J.9(b). After the multiplication, P = 126 and A = 501, with $g = 5$, $r = 0$ and $s = 1$. Since the high-order digit of P is nonzero, case (2) applies and $r := g$, so that $r = 5$, as the arrow indicates in Figure J.9. Since $r = 5$, we could be in a halfway case, but $s = 1$ indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product.

The precise rules for rounding depend on the rounding mode and are given in Figure J.11. Note that P is nonnegative, that is, it contains the magnitude of the result. A good discussion of more efficient ways to implement rounding is in Santoro, Bewick, and Horowitz [1989].

Example In binary with $p = 4$, show how the multiplication algorithm computes the product -5×10 in each of the four rounding modes.

Answer In binary, -5 is $-1.010_2 \times 2^2$ and $10 = 1.010_2 \times 2^3$. Applying the integer multiplication algorithm to the significands gives 01100100_2, so $P = 0110_2$, $A = 0100_2$, $g = 0$, $r = 1$, and $s = 0$. The high-order bit of P is 0, so case (1) applies. Thus, P becomes 1100_2, and since the result is negative, Figure J.11 gives:

round to $-\infty$	1101_2	add 1 since $r \vee s = 1 / 0 = \text{TRUE}$	
round to $+\infty$	1100_2		
round to 0	1100_2		
round to nearest	1100_2	no add since	$r \wedge p_0 = 1 \wedge 0 = \text{FALSE}$ and $r \wedge s = 1 \wedge 0 = \text{FALSE}$

The exponent is $2 + 3 = 5$, so the result is $-1.100_2 \times 2^5 = -48$, except when rounding to $-\infty$, in which case it is $-1.101_2 \times 2^5 = -52$.

Rounding mode	Sign of result ≥ 0	Sign of result < 0
$-\infty$		$+1$ if $r \vee s$
$+\infty$	$+1$ if $r \vee s$	
0		
Nearest	$+1$ if $r \wedge p_0$ or $r \wedge s$	$+1$ if $r \wedge p_0$ or $r \wedge s$

Figure J.11 Rules for implementing the IEEE rounding modes. Let *S* be the magnitude of the preliminary result. Blanks mean that the *p* most-significant bits of *S* are the actual result bits. If the condition listed is true, add 1 to the *p*th most-significant bit of *S*. The symbols *r* and *s* represent the round and sticky bits, while p_0 is the *p*th most-significant bit of *S*.

Overflow occurs when the rounded result is too large to be represented. In single precision, this occurs when the result has an exponent of 128 or higher. If e_1 and e_2 are the two biased exponents, then $1 \leq e_i \leq 254$, and the exponent calculation $e_1 + e_2 - 127$ gives numbers between $1 + 1 - 127$ and $254 + 254 - 127$, or between -125 and 381. This range of numbers can be represented using 9 bits. So one way to detect overflow is to perform the exponent calculations in a 9-bit adder (see Exercise J.12). Remember that you must check for overflow *after* rounding—the example in Figure J.9(c) shows that this can make a difference.

Denormals

Checking for underflow is somewhat more complex because of denormals. In single precision, if the result has an exponent less than -126, that does not necessarily indicate underflow, because the result might be a denormal number. For example, the product of (1×2^{-64}) with (1×2^{-65}) is 1×2^{-129}, and -129 is below the legal exponent limit. But this result is a valid denormal number, namely, 0.125×2^{-126}. In general, when the unbiased exponent of a product dips below -126, the resulting product must be shifted right and the exponent incremented until the exponent reaches -126. If this process causes the entire significand to be shifted out, then underflow has occurred. The precise definition of underflow is somewhat subtle—see Section J.7 for details.

When one of the operands of a multiplication is denormal, its significand will have leading zeros, and so the product of the significands will also have leading zeros. If the exponent of the product is less than -126, then the result is denormal, so right-shift and increment the exponent as before. If the exponent is greater than -126, the result may be a normalized number. In this case, *left*-shift the product (while decrementing the exponent) until either it becomes normalized or the exponent drops to -126.

Denormal numbers present a major stumbling block to implementing floating-point multiplication, because they require performing a variable shift in the multiplier, which wouldn't otherwise be needed. Thus, high-performance, floating-point multipliers often do not handle denormalized numbers, but instead trap, letting software handle them. A few practical codes frequently underflow, even when working properly, and these programs will run quite a bit slower on systems that require denormals to be processed by a trap handler.

So far we haven't mentioned how to deal with operands of zero. This can be handled by either testing both operands before beginning the multiplication or testing the product afterward. If you test afterward, be sure to handle the case $0 \times \infty$ properly: This results in NaN, not 0. Once you detect that the result is 0, set the biased exponent to 0. Don't forget about the sign. The sign of a product is the XOR of the signs of the operands, even when the result is 0.

Precision of Multiplication

In the discussion of integer multiplication, we mentioned that designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a "single times single gets double" instruction. The situation is different for floating point. Many languages allow assigning the product of two single-precision variables to a double-precision one, and the construction can also be exploited by numerical algorithms. The best-known case is using iterative refinement to solve linear systems of equations.

J.5

Floating-Point Addition

Typically, a floating-point operation takes two inputs with p bits of precision and returns a p-bit result. The ideal algorithm would compute this by first performing the operation exactly, and then rounding the result to p bits (using the current rounding mode). The multiplication algorithm presented in the previous section follows this strategy. Even though hardware implementing IEEE arithmetic must return the same result as the ideal algorithm, it doesn't need to actually perform the ideal algorithm. For addition, in fact, there are better ways to proceed. To see this, consider some examples.

First, the sum of the binary 6-bit numbers 1.10011_2 and $1.10001_2 \times 2^{-5}$: When the summands are shifted so they have the same exponent, this is

$$\begin{array}{rl} & 1.10011 \\ + & .0000110001 \\ \hline \end{array}$$

Using a 6-bit adder (and discarding the low-order bits of the second addend) gives

$$\begin{array}{rr} & 1.10011 \\ + & .00001 \\ \hline + & 1.10100 \end{array}$$

The first discarded bit is 1. This isn't enough to decide whether to round up. The rest of the discarded bits, 0001, need to be examined. Or, actually, we just need to record whether any of these bits are nonzero, storing this fact in a sticky bit just as in the multiplication algorithm. So, for adding two p-bit numbers, a p-bit adder is sufficient, as long as the first discarded bit (round) and the OR of the rest of the bits (sticky) are kept. Then Figure J.11 can be used to determine if a roundup is necessary, just as with multiplication. In the example above, sticky is 1, so a roundup is necessary. The final sum is 1.10101_2.

Here's another example:

$$\begin{array}{lr} & 1.11011 \\ + & .0101001 \\ \hline \end{array}$$

A 6-bit adder gives:

$$\begin{array}{lr} & 1.11011 \\ + & .01010 \\ \hline + & 10.00101 \end{array}$$

Because of the carry-out on the left, the round bit isn't the first discarded bit; rather, it is the low-order bit of the sum (1). The discarded bits, 01, are OR'ed together to make sticky. Because round and sticky are both 1, the high-order 6 bits of the sum, 10.0010_2, must be rounded up for the final answer of 10.0011_2.

Next, consider subtraction and the following example:

$$\begin{array}{ll} & 1.00000 \\ - & .00000101111 \\ \hline \end{array}$$

The simplest way of computing this is to convert $-.00000101111_2$ to its two's complement form, so the difference becomes a sum:

$$\begin{array}{ll} & 1.00000 \\ + & 1.11111010001 \end{array}$$

Computing this sum in a 6-bit adder gives:

$$\begin{array}{lr} & 1.00000 \\ + & 1.11111 \\ \hline & 0.11111 \end{array}$$

Because the top bits canceled, the first discarded bit (the guard bit) is needed to fill in the least-significant bit of the sum, which becomes 0.111110_2, and the second discarded bit becomes the round bit. This is analogous to case (1) in the multiplication algorithm (see page J-19). The round bit of 1 isn't enough to decide whether to round up. Instead, we need to OR all the remaining bits (0001) into a sticky bit. In this case, sticky is 1, so the final result must be rounded up to 0.111111. This example shows that if subtraction causes the most-significant bit to cancel, then one guard bit is needed. It is natural to ask whether two guard bits are needed for the case when the *two* most-significant bits cancel. The answer is no, because if x and y are so close that the top two bits of $x-y$ cancel, then $x-y$ will be exact, so guard bits aren't needed at all.

To summarize, addition is more complex than multiplication because, depending on the signs of the operands, it may actually be a subtraction. If it is an addition, there can be carry-out on the left, as in the second example. If it is subtraction, there can be cancellation, as in the third example. In each case, the position of the round bit is different. However, we don't need to compute the exact sum and then round. We can infer it from the sum of the high-order p bits together with the round and sticky bits.

The rest of this section is devoted to a detailed discussion of the floatingpoint addition algorithm. Let a_1 and a_2 be the two numbers to be added. The notations e_i and s_i are used for the exponent and significand of the addends a_i. This means that the floating-point inputs have been unpacked and that s_i has an explicit leading bit. To add a_1 and a_2, perform these eight steps:

1. If $e_1 < e_2$, swap the operands. This ensures that the difference of the exponents satisfies $d = e_1 - e_2 \geq 0$. Tentatively set the exponent of the result to e_1.
2. If the signs of a_1 and a_2 differ, replace s_2 by its two's complement.
3. Place s_2 in a p-bit register and shift it $d = e_1 - e_2$ places to the right (shifting in 1's if s_2 was complemented in the previous step). From the bits shifted out, set g to the most-significant bit, set r to the next most-significant bit, and set sticky to the OR of the rest.
4. Compute a preliminary significand $S = s_1 + s_2$ by adding s_1 to the p-bit register containing s_2. If the signs of a_1 and a_2 are different, the most-significant bit of S is 1, and there was no carry-out, then S is negative. Replace S with its two's complement. This can only happen when $d = 0$.
5. Shift S as follows. If the signs of a_1 and a_2 are the same and there was a carryout in step 4, shift S right by one, filling in the high-order position with 1 (the carry-out). Otherwise, shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the g bit. After that, shift in zeros. Adjust the exponent of the result accordingly.
6. Adjust r and s. If S was shifted right in step 5, set $r :=$ low-order bit of S before shifting and $s := g$ OR r OR s. If there was no shift, set $r := g$, $s := r$ OR s. If there was a single left shift, don't change r and s. If there were two or more left shifts, $r := 0$, $s := 0$. (In the last case, two or more shifts can only happen when a_1 and a_2 have opposite signs and the same exponent, in which case the computation $s_1 + s_2$ in step 4 will be exact.)
7. Round S using Figure J.11; namely, if a table entry is nonempty, add 1 to the low-order bit of S. If rounding causes carry-out, shift S right and adjust the exponent. This is the significand of the result.
8. Compute the sign of the result. If a_1 and a_2 have the same sign, this is the sign of the result. If a_1 and a_2 have different signs, then the sign of the result depends on which of a_1 or a_2 is negative, whether there was a swap in step 1, and whether S was replaced by its two's complement in step 4. See Figure J.12.

swap	compl	sign(a_1)	sign(a_2)	sign(result)
Yes		+	−	−
Yes		−	+	+
No	No	+	−	+
No	No	−	+	−
No	Yes	+	−	−
No	Yes	−	+	+

Figure J.12 Rules for computing the sign of a sum when the addends have different signs. The *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are "don't care."

Example Use the algorithm to compute the sum $(-1.001_2 \times 2^{-2}) + (-1.111_2 \times 2^0)$.

Answer $s_1 = 1.001$, $e_1 = -2$, $s_2 = 1.111$, $e_2 = 0$

1. $e_1 < e_2$, so swap. $d = 2$. Tentative exp $= 0$.
2. Signs of both operands negative, don't negate s_2.
3. Shift s_2 (1.001 after swap) right by 2, giving $s_2 = .010$, $g = 0$, $r = 1$, $s = 0$.
4. $$\begin{array}{r} 1.111 \\ +\quad .010 \\ \hline (1)0.001 \end{array} \qquad S = 0.001, \text{ with a carry-out.}$$
5. Carry-out, so shift S right, $S = 1.000$, exp $=$ exp $+ 1$, so exp $= 1$.
6. $r =$ low-order bit of sum $= 1$, $s = g \vee r \vee s = 0 \vee 1 \vee 0 = 1$.
7. r AND s = TRUE, so Figure J.11 says round up, $S = S + 1$ or $S = 1.001$.
8. Both signs negative, so sign of result is negative. Final answer: $-S \times 2^{\text{exp}} = 1.001_2 \times 2^1$.

Example Use the algorithm to compute the sum $(-1.010_2) + 1.100_2$.

Answer $s_1 = 1.010$, $e_1 = 0$, $s_2 = 1.100$, $e_2 = 0$

1. No swap, $d = 0$, tentative exp $= 0$.
2. Signs differ, replace s_2 with 0.100.
3. $d = 0$, so no shift. $r = g = s = 0$.
4. $$\begin{array}{r} 1.010 \\ +\ 0.100 \\ \hline 1.110 \end{array}$$ Signs are different, most-significant bit is 1, no carry-out, so must two's complement sum, giving $S = 0.010$.

5. Shift left twice, so $S = 1.000$, $\exp = \exp - 2$, or $\exp = -2$.
6. Two left shifts, so $r = g = s = 0$.
7. No addition required for rounding.
8. Answer is $\text{sign} \times S \times 2^{\exp}$ or $\text{sign} \times 1.000 \times 2^{-2}$. Get sign from Figure J.12. Since complement but no swap and $\text{sign}(a_1)$ is $-$, the sign of the sum is $+$. Thus, the answer $= 1.000_2 \times 2^{-2}$.

Speeding Up Addition

Let's estimate how long it takes to perform the algorithm above. Step 2 may require an addition, step 4 requires one or two additions, and step 7 may require an addition. If it takes T time units to perform a p-bit add (where $p = 24$ for single precision, 53 for double), then it appears the algorithm will take at least 4 T time units. But that is too pessimistic. If step 4 requires two adds, then a_1 and a_2 have the same exponent and different signs, but in that case the difference is exact, so no roundup is required in step 7. Thus, only three additions will ever occur. Similarly, it appears that a variable shift may be required both in step 3 and step 5. But if $|e_1 - e_2| \leq 1$, then step 3 requires a right shift of at most one place, so only step 5 needs a variable shift. And, if $|e_1 - e_2| > 1$, then step 3 needs a variable shift, but step 5 will require a left shift of at most one place. So only a single variable shift will be performed. Still, the algorithm requires three sequential adds, which, in the case of a 53-bit double-precision significand, can be rather time consuming.

A number of techniques can speed up addition. One is to use pipelining. The "Putting It All Together" section gives examples of how some commercial chips pipeline addition. Another method (used on the Intel 860 [Kohn and Fu 1989]) is to perform two additions in parallel. We now explain how this reduces the latency from $3T$ to T.

There are three cases to consider. First, suppose that both operands have the same sign. We want to combine the addition operations from steps 4 and 7. The position of the high-order bit of the sum is not known ahead of time, because the addition in step 4 may or may not cause a carry-out. Both possibilities are accounted for by having two adders. The first adder assumes the add in step 4 will not result in a carry-out. Thus, the values of r and s can be computed before the add is actually done. If r and s indicate that a roundup is necessary, the first adder will compute $S = s_1 + s_2 + 1$, where the notation $+1$ means adding 1 at the position of the least-significant bit of s_1. This can be done with a regular adder by setting the low-order carry-in bit to 1. If r and s indicate no roundup, the adder computes $S = s_1 + s_2$ as usual. One extra detail: When $r = 1$, $s = 0$, you will also need to know the low-order bit of the sum, which can also be computed in advance very quickly. The second adder covers the possibility that there will be carry-out. The values of r and s and the position where the roundup 1 is added are different from above, but again they can be quickly computed in advance. It is not known whether there will be a carry-out until after the add is actually done, but that doesn't matter. By doing both adds in parallel, one adder is guaranteed to reduce the correct answer.

The next case is when a_1 and a_2 have opposite signs but the same exponent. The sum $a_1 + a_2$ is exact in this case (no roundup is necessary) but the sign isn't known until the add is completed. So don't compute the two's complement (which requires an add) in step 2, but instead compute $\overline{s}_1 + s_2 + 1$ and $s_1 + \overline{s}_2 + 1$ in parallel. The first sum has the result of simultaneously complementing s_1 and computing the sum, resulting in $s_2 - s_1$. The second sum computes $s_1 - s_2$. One of these will be nonnegative and hence the correct final answer. Once again, all the additions are done in one step using two adders operating in parallel.

The last case, when a_1 and a_2 have opposite signs and different exponents, is more complex. If $|e_1 - e_2| > 1$, the location of the leading bit of the difference is in one of two locations, so there are two cases just as in addition. When $|e_1 - e_2| = 1$, cancellation is possible and the leading bit could be almost anywhere. However, only if the leading bit of the difference is in the same position as the leading bit of s_1 could a roundup be necessary. So one adder assumes a roundup, and the other assumes no roundup. Thus, the addition of step 4 and the rounding of step 7 can be combined. However, there is still the problem of the addition in step 2!

To eliminate this addition, consider the following diagram of step 4:

$$\begin{array}{ll} & |\!-\!-\; p \;-\!-\!| \\ s_1 & 1.xxxxxxx \\ s_2 - & \qquad\quad\;\; 1\underline{\underline{yyzzzzz}} \end{array}$$

If the bits marked z are all 0, then the high-order p bits of $S = s_1 - s_2$ can be computed as $s_1 + \overline{s}_2 + 1$. If at least one of the z bits is 1, use $s_1 + \overline{s}_2$. So $s_1 - s_2$ can be computed with one addition. However, we still don't know g and r for the two's complement of s_2, which are needed for rounding in step 7.

To compute $s_1 - s_2$ and get the proper g and r bits, combine steps 2 and 4 as follows. Don't complement s_2 in step 2. Extend the adder used for computing S two bits to the right (call the extended sum S'). If the preliminary sticky bit (computed in step 3) is 1, compute $S' = s_1' + \overline{s}_2'$, where s_1' has two 0 bits tacked onto the right, and s_2' has preliminary g and r appended. If the sticky bit is 0, compute $s_1' + \overline{s}_2' + 1$. Now the two low-order bits of S' have the correct values of g and r (the sticky bit was already computed properly in step 3). Finally, this modification can be combined with the modification that combines the addition from steps 4 and 7 to provide the final result in time T, the time for one addition.

A few more details need to be considered, as discussed in Santoro, Bewick, and Horowitz [1989] and Exercise J.17. Although the Santoro paper is aimed at multiplication, much of the discussion applies to addition as well. Also relevant is Exercise J.19, which contains an alternative method for adding signed magnitude numbers.

Denormalized Numbers

Unlike multiplication, for addition very little changes in the preceding description if one of the inputs is a denormal number. There must be a test to see if the exponent field is 0. If it is, then when unpacking the significand there will not be a leading 1.

By setting the biased exponent to 1 when unpacking a denormal, the algorithm works unchanged.

To deal with denormalized outputs, step 5 must be modified slightly. Shift S until it is normalized, or until the exponent becomes $E_{\min}$ (that is, the biased exponent becomes 1). If the exponent is $E_{\min}$ and, after rounding, the high-order bit of S is 1, then the result is a normalized number and should be packed in the usual way, by omitting the 1. If, on the other hand, the high-order bit is 0, the result is denormal. When the result is unpacked, the exponent field must be set to 0. Section J.7 discusses the exact rules for detecting underflow.

Incidentally, detecting overflow is very easy. It can only happen if step 5 involves a shift right and the biased exponent at that point is bumped up to 255 in single precision (or 2047 for double precision), or if this occurs after rounding.

J.6 Division and Remainder

In this section, we'll discuss floating-point division and remainder.

Iterative Division

We earlier discussed an algorithm for integer division. Converting it into a floating-point division algorithm is similar to converting the integer multiplication algorithm into floating point. The formula

$$(\mathrm{s}_1 \times 2^{e_1})/(\mathrm{s}_2 \times 2^{e_2}) = (\mathrm{s}_1/\mathrm{s}_2) \times 2^{e_1 - e_2}$$

shows that if the divider computes s_1/s_2, then the final answer will be this quotient multiplied by $2^{e_1-e_2}$. Referring to Figure J.2(b) (page J-4), the alignment of operands is slightly different from integer division. Load s_2 into B and s_1 into P. The A register is not needed to hold the operands. Then the integer algorithm for division (with the one small change of skipping the very first left shift) can be used, and the result will be of the form $q_0 . q_1 \cdots$. To round, simply compute two additional quotient bits (guard and round) and use the remainder as the sticky bit. The guard digit is necessary because the first quotient bit might be 0. However, since the numerator and denominator are both normalized, it is not possible for the two most-significant quotient bits to be 0. This algorithm produces one quotient bit in each step.

A different approach to division converges to the quotient at a quadratic rather than a linear rate. An actual machine that uses this algorithm will be discussed in Section J.10. First, we will describe the two main iterative algorithms, and then we will discuss the pros and cons of iteration when compared with the direct algorithms. A general technique for constructing iterative algorithms, called *Newton's iteration*, is shown in Figure J.13. First, cast the problem in the form of finding the zero of a function. Then, starting from a guess for the zero, approximate the function by its tangent at that guess and form a new guess based

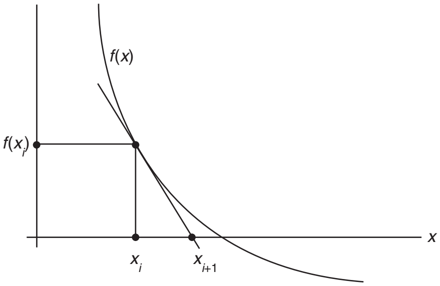


Figure J.13 Newton's iteration for zero finding. If x_i is an estimate for a zero of *f*, then x_{i+1} is a better estimate. To compute x_{i+1}, find the intersection of the *x*-axis with the tangent line to *f* at $f(x_i)$.

on where the tangent has a zero. If x_i is a guess at a zero, then the tangent line has the equation:

$$y - f(x_i) = f'(x_i)(x - x_i)$$

This equation has a zero at

J.6.1
$$x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

To recast division as finding the zero of a function, consider $f(x) = x^{-1} - b$. Since the zero of this function is at $1/b$, applying Newton's iteration to it will give an iterative method of computing $1/b$ from b. Using $f'(x) = -1/x^2$, Equation J.6.1 becomes:

J.6.2
$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

Thus, we could implement computation of a/b using the following method:

1. Scale b to lie in the range $1 \le b < 2$ and get an approximate value of $1/b$ (call it x_0) using a table lookup.
2. Iterate $x_{i+1} = x_i(2 - x_i b)$ until reaching an x_n that is accurate enough.
3. Compute ax_n and reverse the scaling done in step 1.

Here are some more details. How many times will step 2 have to be iterated? To say that x_i is accurate to p bits means that $|(x_i - 1/b)/(1/b)| = 2^{-p}$, and a simple algebraic manipulation shows that when this is so, then $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$. Thus, the number of correct bits doubles at each step. Newton's iteration is *self-correcting* in the sense that making an error in x_i doesn't really matter. That is, it treats x_i as a guess at $1/b$ and returns x_{i+1} as an improvement on it (roughly doubling the digits). One thing that would cause x_i to be in error is rounding error. More importantly, however, in the early iterations we can take advantage of the fact that we don't expect many correct bits by performing the multiplication in reduced precision, thus gaining speed without sacrificing accuracy. Another application of Newton's iteration is discussed in Exercise J.20.

The second iterative division method is sometimes called *Goldschmidt's algorithm*. It is based on the idea that to compute a/b, you should multiply the numerator and denominator by a number r with $rb \approx 1$. In more detail, let $x_0 = a$ and $y_0\text{þ} = b$. At each step compute $x_{i+1} = r_i x_i$ and $y_{i+1} = r_i y_i$. Then the quotient $x_{i+1}/y_{i+1} = x_i/y_i = a/b$ is constant. If we pick r_i so that $y_i \rightarrow 1$, then $x_i \rightarrow a/b$, so the x_i converge to the answer we want. This same idea can be used to compute other functions. For example, to compute the square root of a, let $x_0 = a$ and $y_0 = a$, and at each step compute $x_{i+1} = r_i^2 x_i$, $y_{i+1} = r_i y_i$. Then $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$, so if the r_i are chosen to drive $x_i \rightarrow 1$, then $y_i \rightarrow \sqrt{a}$. This technique is used to compute square roots on the TI 8847.

Returning to Goldschmidt's division algorithm, set $x_0 = a$ and $y_0 = b$, and write $b = 1 - \delta$, where $|\delta| < 1$. If we pick $r_0 = 1 + \delta$, then $y_1 = r_0 y_0 = 1 - \delta^2$. We next pick $r_1 = 1 + \delta^2$, so that $y_2 = r_1 y_1 = 1 - \delta^4$, and so on. Since $|\delta| < 1$, $y_i \rightarrow 1$. With this choice of r_i, the x_i will be computed as $x_{i+1} = r_i x_i = \left(1 + \delta^{2^i}\right) x_i = \left(1 + (1-b)^{2^i}\right) x_i$, or

$$x_{i+1} = a[1 + (1-b)]\left[1 + (1-b)^2\right]\left[1 + (1-b)^4\right] \cdots \left[1 + (1-b)^{2^i}\right] \tag{J.6.3}$$

There appear to be two problems with this algorithm. First, convergence is slow when b is not near 1 (that is, δ is not near 0), and, second, the formula isn't self-correcting—since the quotient is being computed as a product of independent terms, an error in one of them won't get corrected. To deal with slow convergence, if you want to compute a/b, look up an approximate inverse to b (call it b'), and run the algorithm on ab'/bb'. This will converge rapidly since $bb' \approx 1$.

To deal with the self-correction problem, the computation should be run with a few bits of extra precision to compensate for rounding errors. However, Goldschmidt's algorithm does have a weak form of self-correction, in that the precise value of the r_i does not matter. Thus, in the first few iterations, when the full precision of $1 - \delta^{2^i}$ is not needed you can choose r_i to be a truncation of $1 + \delta^{2^i}$, which may make these iterations run faster without affecting the speed of convergence. If r_i is truncated, then y_i is no longer exactly $1 - \delta^{2^i}$. Thus, Equation J.6.3 can no longer be used, but it is easy to organize the computation so that it does not depend on the precise value of r_i. With these changes, Goldschmidt's algorithm is as follows (the notes in brackets show the connection with our earlier formulas).

1. Scale a and b so that $1 \leq b < 2$.
2. Look up an approximation to $1/b$ (call it b') in a table.
3. Set $x_0 = ab'$ and $y_0 = bb'$.

4. Iterate until x_i is close enough to a/b:

```
Loop
        $r \approx 2 - y$              $[\text{if } y_i = 1 + \delta_i, \text{then } r \approx 1 - \delta_i]$
        $y = y \times r$               $[y_{i+1} = y_i \times r \approx 1 - \delta_i^2]$
        $x_{i+1} = x_i \times r$       $[x_{i+1} = x_i \times r]$
End loop
```

The two iteration methods are related. Suppose in Newton's method that we unroll the iteration and compute each term x_{i+1} directly in terms of b, instead of recursively in terms of x_i. By carrying out this calculation (see Exercise J.22), we discover that

$$x_{i+1} = x_0(2 - x_0 b)\Big[\Big(1 + (x_0 b - 1)^2\Big]\Big[1 + (x_0 b - 1)^4\Big]\cdots\Big[1 + (x_0 b - 1)^{2^i}\Big]$$

This formula is very similar to Equation J.6.3. In fact, they are identical if a and b in J.6.3 are replaced with ax_0, bx_0, and $a = 1$. Thus, if the iterations were done to infinite precision, the two methods would yield exactly the same sequence x_i.

The advantage of iteration is that it doesn't require special divide hardware. Instead, it can use the multiplier (which, however, requires extra control). Further, on each step, it delivers twice as many digits as in the previous step—unlike ordinary division, which produces a fixed number of digits at every step.

There are two disadvantages with inverting by iteration. The first is that the IEEE standard requires division to be correctly rounded, but iteration only delivers a result that is close to the correctly rounded answer. In the case of Newton's iteration, which computes $1/b$ instead of a/b directly, there is an additional problem. Even if $1/b$ were correctly rounded, there is no guarantee that a/b will be. An example in decimal with $p = 2$ is $a = 13$, $b = 51$. Then $a/b = .2549\ldots$, which rounds to .25. But $1/b = .0196\ldots$, which rounds to .020, and then $a \times .020 = .26$, which is off by 1. The second disadvantage is that iteration does not give a remainder. This is especially troublesome if the floating-point divide hardware is being used to perform integer division, since a remainder operation is present in almost every high-level language.

Traditional folklore has held that the way to get a correctly rounded result from iteration is to compute $1/b$ to slightly more than $2p$ bits, compute a/b to slightly more than $2p$ bits, and then round to p bits. However, there is a faster way, which apparently was first implemented on the TI 8847. In this method, a/b is computed to about 6 extra bits of precision, giving a preliminary quotient q. By comparing qb with a (again with only 6 extra bits), it is possible to quickly decide whether q is correctly rounded or whether it needs to be bumped up or down by 1 in the least-significant place. This algorithm is explored further in Exercise J.21.

One factor to take into account when deciding on division algorithms is the relative speed of division and multiplication. Since division is more complex than multiplication, it will run more slowly. A common rule of thumb is that division algorithms should try to achieve a speed that is about one-third that of multiplication.

One argument in favor of this rule is that there are real programs (such as some versions of spice) where the ratio of division to multiplication is 1:3. Another place where a factor of 3 arises is in the standard iterative method for computing square root. This method involves one division per iteration, but it can be replaced by one using three multiplications. This is discussed in Exercise J.20.

Floating-Point Remainder

For nonnegative integers, integer division and remainder satisfy:

$$a = (a \text{ DIV } b)b + a \text{ REM } b, \qquad 0 \leq a \text{ REM } b < b$$

A floating-point remainder x REM y can be similarly defined as $x = \text{INT}(x/y)y + x$ REM y. How should x/y be converted to an integer? The IEEE remainder function uses the round-to-even rule. That is, pick $n = \text{INT}\ (x/y)$ so that $|x/y - n| \leq 1/2$. If two different n satisfy this relation, pick the even one. Then REM is defined to be $x - yn$. Unlike integers where $0 \leq a$ REM $b < b$, for floating-point numbers $|x \text{ REM } y| \leq y/2$. Although this defines REM precisely, it is not a practical operational definition, because n can be huge. In single precision, n could be as large as $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$.

There is a natural way to compute REM if a direct division algorithm is used. Proceed as if you were computing x/y. If $x = s_1 2^{e_1}$ and $y = s_2 2^{e_2}$ and the divider is as in Figure J.2(b) (page J-4), then load s_1 into P and s_2 into B. After $e_1 - e_2$ division steps, the P register will hold a number r of the form $x - yn$ satisfying $0 \leq r < y$. Since the IEEE remainder satisfies $|\text{REM}| \leq y/2$, REM is equal to either r or $r - y$. It is only necessary to keep track of the last quotient bit produced, which is needed to resolve halfway cases. Unfortunately, $e_1 - e_2$ can be a lot of steps, and floating-point units typically have a maximum amount of time they are allowed to spend on one instruction. Thus, it is usually not possible to implement REM directly. None of the chips discussed in Section J.10 implements REM, but they could by providing a remainder-step instruction—this is what is done on the Intel 8087 family. A remainder step takes as arguments two numbers x and y, and performs divide steps until either the remainder is in P or n steps have been performed, where n is a small number, such as the number of steps required for division in the highest-supported precision. Then REM can be implemented as a software routine that calls the REM step instruction $\lfloor (e_1 - e_2)/n \rfloor$ times, initially using x as the numerator but then replacing it with the remainder from the previous REM step.

REM can be used for computing trigonometric functions. To simplify things, imagine that we are working in base 10 with five significant figures, and consider computing sin x. Suppose that $x = 7$. Then we can reduce by $\pi = 3.1416$ and compute $\sin(7) = \sin(7 - 2 \times 3.1416) = \sin(0.7168)$ instead. But, suppose we want to compute $\sin(2.0 \times 10^5)$. Then $2 \times 10^5/3.1416 = 63661.8$, which in our five-place system comes out to be 63662. Since multiplying 3.1416 times 63662 gives 200000.5392, which rounds to 2.0000×10^5, argument reduction reduces 2×10^5 to 0, which is not even close to being correct. The problem is that our five-place system does not have the precision to do correct argument reduction. Suppose we had the REM operator. Then we could compute 2×10^5 REM 3.1416 and get $-.53920$. However, this is still not correct because we used 3.1416, which is an approximation for π. The value of 2×10^5 REM π is $-.071513$.

Traditionally, there have been two approaches to computing periodic functions with large arguments. The first is to return an error for their value when x is large. The second is to store π to a very large number of places and do exact argument reduction. The REM operator is not much help in either of these situations. There is a third approach that has been used in some math libraries, such as the Berkeley UNIX 4.3bsd release. In these libraries, π is computed to the nearest floating-point number. Let's call this machine π, and denote it by π'. Then, when computing sin x, reduce x using x REM π'. As we saw in the above example, x REM π' is quite different from x REM π when x is large, so that computing sin x as sin(x REM π') will not give the exact value of sin x. However, computing trigonometric functions in this fashion has the property that all familiar identities (such as $\sin^2 x + \cos^2 x = 1$) are true to within a few rounding errors. Thus, using REM together with machine π provides a simple method of computing trigonometric functions that is accurate for small arguments and still may be useful for large arguments.

When REM is used for argument reduction, it is very handy if it also returns the low-order bits of n (where x REM $y = x - ny$). This is because a practical implementation of trigonometric functions will reduce by something smaller than 2π. For example, it might use $\pi/2$, exploiting identities such as $\sin(x - \pi/2) = -\cos x$, $\sin(x - \pi) = -\sin x$. Then the low bits of n are needed to choose the correct identity.

J.7 More on Floating-Point Arithmetic

Before leaving the subject of floating-point arithmetic, we present a few additional topics.

Fused Multiply-Add

Probably the most common use of floating-point units is performing matrix operations, and the most frequent matrix operation is multiplying a matrix times a matrix (or vector), which boils down to computing an inner product, $x_1 \cdot y_1 + x_2 \cdot y_2 + \ldots + x_n \cdot y_n$. Computing this requires a series of multiply-add combinations.

Motivated by this, the IBM RS/6000 introduced a single instruction that computes $ab + c$, the *fused multiply-add*. Although this requires being able to read three operands in a single instruction, it has the potential for improving the performance of computing inner products.

The fused multiply-add computes $ab + c$ exactly and then rounds. Although rounding only once increases the accuracy of inner products somewhat, that is not its primary motivation. There are two main advantages of rounding once. First, as we saw in the previous sections, rounding is expensive to implement because it may require an addition. By rounding only once, an addition operation has been eliminated. Second, the extra accuracy of fused multiply-add can be used to compute correctly rounded division and square root when these are not available directly in hardware. Fused multiply-add can also be used to implement efficient floating-point multiple-precision packages.

The implementation of correctly rounded division using fused multiply-add has many details, but the main idea is simple. Consider again the example from Section J.6 (page J-30), which was computing a/b with $a=13$, $b=51$. Then $1/b$ rounds to $b'=.020$, and ab' rounds to $q'=.26$, which is not the correctly rounded quotient. Applying fused multiply-add twice will correctly adjust the result, via the formulas

$$r = a - bq'$$

$$q'' = q' + rb'$$

Computing to two-digit accuracy, $bq'=51\times.26$ rounds to 13, and so $r=a-bq'$ would be 0, giving no adjustment. But using fused multiply-add gives $r=a-bq'=13-(51\times.26)=-.26$, and then $q''=q'+rb'=.26-.0052=.2548$, which rounds to the correct quotient, .25. More details can be found in the papers by Montoye, Hokenek, and Runyon [1990] and Markstein [1990].

Precisions

The standard specifies four precisions: *single*, *single extended*, *double*, and *double extended*. The properties of these precisions are summarized in Figure J.7 (page J-16). Implementations are not required to have all four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented. Thus, the computer designer usually only has to decide whether to support double extended and, if so, how many bits it should have.

The Motorola 68882 and Intel 387 coprocessors implement extended precision using the smallest allowable size of 80 bits (64 bits of significand). However, many of the more recently designed, high-performance floating-point chips do not implement 80-bit extended precision. One reason is that the 80-bit width of extended precision is awkward for 64-bit buses and registers. Some new architectures, such as SPARC V8 and PA-RISC, specify a 128-bit extended (or *quad*) precision. They have established a *de facto* convention for quad that has 15 bits of exponent and 113 bits of significand.

Although most high-level languages do not provide access to extended precision, it is very useful to writers of mathematical software. As an example, consider writing a library routine to compute the length of a vector (x,y) in the plane, namely, $\sqrt{x^2+y^2}$. If x is larger than $2^{E_{\max}/2}$, then computing this in the obvious way will overflow. This means that either the allowable exponent range for this subroutine will be cut in half or a more complex algorithm using scaling will have to be employed. But, if extended precision is available, then the simple algorithm will work. Computing the length of a vector is a simple task, and it is not difficult to come up with an algorithm that doesn't overflow. However, there are more complex problems for which extended precision means the difference between a simple, fast algorithm and a much more complex one. One of the best examples of this is binary-to-decimal conversion. An efficient algorithm for binary-to-decimal conversion that makes essential use of extended precision is very readably presented in Coonen [1984]. This algorithm is also briefly sketched in Goldberg [1991]. Computing accurate values for transcendental functions is another example of a problem that is made much easier if extended precision is present.

One very important fact about precision concerns *double rounding*. To illustrate in decimals, suppose that we want to compute 1.9×0.66 and that single precision is two digits, while extended precision is three digits. The exact result of the product is 1.254. Rounded to extended precision, the result is 1.25. When further rounded to single precision, we get 1.2. However, the result of 1.9×0.66 correctly rounded to single precision is 1.3. Thus, rounding twice may not produce the same result as rounding once. Suppose you want to build hardware that only does double-precision arithmetic. Can you simulate single precision by computing first in double precision and then rounding to single? The above example suggests that you can't. However, double rounding is not always dangerous. In fact, the following rule is true (this is not easy to prove, but see Exercise J.25).

If x *and* y *have* p*-bit significands, and* $\mathrm{x}+\mathrm{y}$ *is computed exactly and then rounded to* q *places, a second rounding to* p *places will not change the answer if* $\mathrm{q} \geq 2\mathrm{p}+2$*. This is true not only for addition, but also for multiplication, division, and square root.*

In our example above, $q=3$ and $p=2$, so q Š $2p+2$ is not true. On the other hand, for IEEE arithmetic, double precision has $q=53$ and $p=24$, so $q=53$ Š $2p+2=50$. Thus, single precision can be implemented by computing in double precision—that is, computing the answer exactly and then rounding to double—and then rounding to single precision.

Exceptions

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact, and invalid. By default, when these exceptions occur, they merely set a flag and the computation continues. The flags are *sticky*, meaning that once set they remain set until explicitly cleared. The standard strongly encourages implementations to provide a trap-enable bit for each exception. When an exception with an enabled trap handler occurs, a user trap handler is called, and the value of the associated exception flag is undefined. In Section J.3 we mentioned that $\sqrt{-3}$ has the value NaN and $1/0$ is ∞. These are examples of operations that raise an exception.

By default, computing $\sqrt{-3}$ sets the invalid flag and returns the value NaN. Similarly 1/0 sets the divide-by-zero flag and returns ∞.

The underflow, overflow, and divide-by-zero exceptions are found in most other systems. The *invalid exception* is for the result of operations such as $\sqrt{-1}$, 0/0, or $\infty - \infty$, which don't have any natural value as a floating-point number or as $\pm\infty$. The *inexact exception* is peculiar to IEEE arithmetic and occurs either when the result of an operation must be rounded or when it overflows. In fact, since 1/0 and an operation that overflows both deliver ∞, the exception flags must be consulted to distinguish between them. The inexact exception is an unusual "exception," in that it is not really an exceptional condition because it occurs so frequently. Thus, enabling a trap handler for the inexact exception will most likely have a severe impact on performance. Enabling a trap handler doesn't affect whether an operation is exceptional except in the case of underflow. This is discussed below.

The IEEE standard assumes that when a trap occurs, it is possible to identify the operation that trapped and its operands. On machines with pipelining or multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support may be necessary to identify exactly which operation trapped.

Another problem is illustrated by the following program fragment.

```
r1 = r2/r3
r2 = r4 + r5
```

These two instructions might well be executed in parallel. If the divide traps, its argument `r2` could already have been overwritten by the addition, especially since addition is almost always faster than division. Computer systems that support trapping in the IEEE standard must provide some way to save the value of `r2`, either in hardware or by having the compiler avoid such a situation in the first place. This kind of problem is not peculiar to floating point. In the sequence

```
r1 = 0(r2)
r2 = r3
```

it would be efficient to execute `r2 = r3` while waiting for memory. But, if accessing `0(r2)` causes a page fault, `r2` might no longer be available for restarting the instruction `r1 = 0(r2)`.

One approach to this problem, used in the MIPS R3010, is to identify instructions that may cause an exception early in the instruction cycle. For example, an addition can overflow only if one of the operands has an exponent of $E_{\max}$, and so on. This early check is conservative: It might flag an operation that doesn't actually cause an exception. However, if such false positives are rare, then this technique will have excellent performance. When an instruction is tagged as being possibly exceptional, special code in a trap handler can compute it without destroying any state. Remember that all these problems occur only when trap handlers are enabled. Otherwise, setting the exception flags during normal processing is straightforward.

Underflow

We have alluded several times to the fact that detection of underflow is more complex than for the other exceptions. The IEEE standard specifies that if an underflow trap handler is enabled, the system must trap if the result is denormal. On the other hand, if trap handlers are disabled, then the underflow flag is set only if there is a loss of accuracy—that is, if the result must be rounded. The rationale is, if no accuracy is lost on an underflow, there is no point in setting a warning flag. But if a trap handler is enabled, the user might be trying to simulate flush-to-zero and should therefore be notified whenever a result dips below $1.0 \times 2^{E_{\min}}$.

So if there is no trap handler, the underflow exception is signaled only when the result is denormal and inexact, but the definitions of *denormal* and *inexact* are both subject to multiple interpretations. Normally, inexact means there was a result that couldn't be represented exactly and had to be rounded. Consider the example (in a base 2 floating-point system with 3-bit significands) of $\left(1.11_2 \times 2^{-2}\right) \times \left(1.11_2 \times 2^{E_{\min}}\right) = 0.110001_2 \times 2^{E_{\min}}$, with round to nearest in effect. The delivered result is $0.11_2 \times 2^{E_{\min}}$, which had to be rounded, causing inexact to be signaled. But is it correct to also signal underflow? Gradual underflow loses significance because the exponent range is bounded. If the exponent range were unbounded, the delivered result would be $1.10_2 \times 2^{E_{\min}-1}$, exactly the same answer obtained with gradual underflow. The fact that denormalized numbers have fewer bits in their significand than normalized numbers therefore doesn't make any difference in this case. The commentary to the standard [Cody et al. 1984] encourages this as the criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same fraction size, but with a very large exponent range. However, owing to the difficulty of implementing this scheme, the standard allows setting the underflow flag whenever the result is denormal and different from the infinitely precise result.

There are two possible definitions of what it means for a result to be denormal. Consider the example of $1.10_2 \times 2^{-1}$ multiplied by $1.10_2 \times 2^{E_{\min}}$. The exact product is $0.1111 \times 2^{E_{\min}}$. The rounded result is the normal number $1.00_2 \times 2^{E_{\min}}$. Should underflow be signaled? Signaling underflow means that you are using the *before rounding* rule, because the result was denormal before rounding. Not signaling underflow means that you are using the *after rounding* rule, because the result is normalized after rounding. The IEEE standard provides for choosing either rule; however, the one chosen must be used consistently for all operations.

To illustrate these rules, consider floating-point addition. When the result of an addition (or subtraction) is denormal, it is always exact. Thus, the underflow flag never needs to be set for addition. That's because if traps are not enabled then no exception is raised. And if traps are enabled, the value of the underflow flag is undefined, so again it doesn't need to be set.

One final subtlety should be mentioned concerning underflow. When there is no underflow trap handler, the result of an operation on p-bit numbers that causes an underflow is a denormal number with $p-1$ or fewer bits of precision. When traps are enabled, the trap handler is provided with the result of the operation rounded to p bits and with the exponent wrapped around. Now there is a potential double-rounding problem. If the trap handler wants to return the denormal result, it can't just round its argument, because that might lead to a double-rounding error. Thus, the trap handler must be passed at least one extra bit of information if it is to be able to deliver the correctly rounded result.

J.8 Speeding Up Integer Addition

The previous section showed that many steps go into implementing floating-point operations; however, each floating-point operation eventually reduces to an integer operation. Thus, increasing the speed of integer operations will also lead to faster floating point.

Integer addition is the simplest operation and the most important. Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to calculate addresses. Despite the simplicity of addition, there isn't a single best way to perform high-speed addition. We will discuss three techniques that are in current use: carry-lookahead, carry-skip, and carry-select.

Carry-Lookahead

An n-bit adder is just a combinational circuit. It can therefore be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. How do you figure out what this circuit looks like? From Equation J.2.1 (page J-3) the formula for the ith sum can be written as:

J.8.1
$$s_i = a_i\overline{b}_i\overline{c}_i + \overline{a}_i b_i \overline{c}_i + \overline{a}_i\overline{b}_i c_i + a_i b_i c_i$$

where c_i is both the carry-in to the ith adder and the carry-out from the $(i-1)$-st adder.

The problem with this formula is that, although we know the values of a_i and b_i—they are inputs to the circuit—we don't know c_i. So our goal is to write c_i in terms of a_i and b_i. To accomplish this, we first rewrite Equation J.2.2 (page J-3) as:

J.8.2
$$c_i = g_{i-1} + p_{i-1}c_{i-1}, \quad g_{i-1} = a_{i-1}b_{i-1}, \quad p_{i-1} = a_{i-1} + b_{i-1}$$

Here is the reason for the symbols p and g: If g_{i-1} is true, then c_i is certainly true, so a carry is *generated*. Thus, g is for generate. If p_{i-1} is true, then if c_{i-1} is true, it is *propagated* to c_i. Start with Equation J.8.1 and use Equation J.8.2 to replace c_i with $g_{i-1}+p_{i-1}c_{i-1}$. Then, use Equation J.8.2 with $i-1$ in place of i to replace c_{i-1} with c_{i-2}, and so on. This gives the result:

J.8.3
$$c_i = g_{i-1} + p_{i-1}g_{i-2} + p_{i-1}p_{i-2}g_{i-3} + \cdots + p_{i-1}p_{i-2}\cdots p_1 g_0 + p_{i-1}p_{i-2}\cdots p_1 p_0 c_0$$

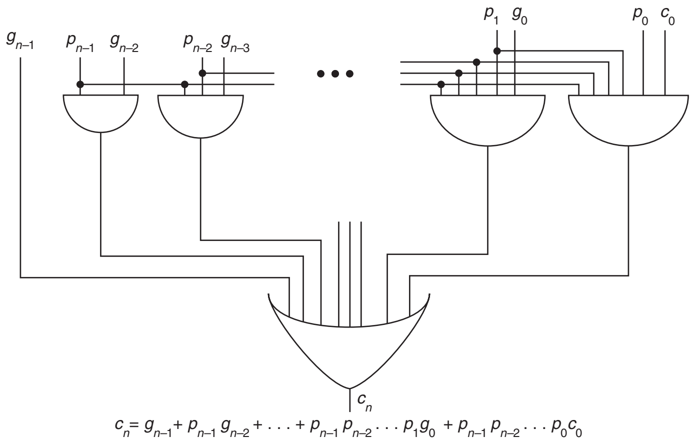

g_{n-1}
p_{n-1}
g_{n-2}
p_{n-2}
g_{n-3}
p_1
g_0
p_0
c_0
c_n
$c_n = g_{n-1} + p_{n-1}\, g_{n-2} + \ldots + p_{n-1}\, p_{n-2} \cdots p_1 g_0 + p_{n-1}\, p_{n-2} \cdots p_0 c_0$

Figure J.14 Pure carry-lookahead circuit for computing the carry-out c_n of an *n*-bit adder.

An adder that computes carries using Equation J.8.3 is called a *carry-lookahead adder*, or CLA. A CLA requires one logic level to form p and g, two levels to form the carries, and two for the sum, for a grand total of five logic levels. This is a vast improvement over the $2n$ levels required for the ripple-carry adder.

Unfortunately, as is evident from Equation J.8.3 or from Figure J.14, a carry-lookahead adder on n bits requires a fan-in of $n+1$ at the OR gate as well as at the rightmost AND gate. Also, the p_{n-1} signal must drive n AND gates. In addition, the rather irregular structure and many long wires of Figure J.14 make it impractical to build a full carry-lookahead adder when n is large.

However, we can use the carry-lookahead idea to build an adder that has about $\log_2 n$ logic levels (substantially fewer than the $2n$ required by a ripplecarry adder) and yet has a simple, regular structure. The idea is to build up the p's and g's in steps. We have already seen that

$$c_1 = g_0 + c_0 p_0$$

This says there is a carry-out of the 0th position (c_1) either if there is a carry generated in the 0th position or if there is a carry into the 0th position and the carry propagates. Similarly,

$$c_2 = G_{01} + P_{01} c_0$$

G_{01} means there is a carry generated out of the block consisting of the first two bits. P_{01} means that a carry propagates through this block. P and G have the following logic equations:

$$G_{01} = g_1 + p_1 g_0$$
$$p_{01} = p_1 p_0$$

More generally, for any j with $i < j$, $j + 1 < k$, we have the recursive relations:

J.8.4 $$c_{k+1} = G_{ik} + P_{ik}c_i$$

J.8.5 $$G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$$

J.8.6 $$P_{ik} = P_{ij}P_{j+1,k}$$

Equation J.8.5 says that a carry is generated out of the block consisting of bits i through k inclusive if it is generated in the high-order part of the block $(j+1,\ k)$ or if it is generated in the low-order part of the block (i,j) and then propagated through the high part. These equations will also hold for $i \le j < k$ if we set $G_{ii} = g_i$ and $P_{ii} = p_i$.

Example Express P_{03} and G_{03} in terms of p's and g's.

Answer Using Equation J.8.6, $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$. Since $P_{ii} = p_i$, $P_{03} = p_0p_1p_2p_3$. For G_{03}, Equation J.8.5 says $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$.

With these preliminaries out of the way, we can now show the design of a practical CLA. The adder consists of two parts. The first part computes various values of P and G from p_i and g_i, using Equations J.8.5 and J.8.6; the second part uses these P and G values to compute all the carries via Equation J.8.4. The first part of the design is shown in Figure J.15. At the top of the diagram, input numbers $a_7 \ldots a_0$ and $b_7 \ldots b_0$ are converted to p's and g's using cells of type 1. Then various P's and G's are generated by combining cells of type 2 in a binary tree structure. The second part of the design is shown in Figure J.16. By feeding c_0 in at the bottom of this tree, all the carry bits come out at the top. Each cell must know a pair of (P,G) values in order to do the conversion, and the value it needs is written inside the cells. Now compare Figures J.15 and J.16. There is a one-to-one correspondence between cells, and the value of (P,G) needed by the carry-generating cells is exactly the value known by the corresponding (P,G)-generating cells. The combined cell is shown in Figure J.17. The numbers to be added flow into the top and downward through the tree, combining with c_0 at the bottom and flowing back up the tree to form the carries. Note that one thing is missing from Figure J.17: a small piece of extra logic to compute c_8 for the carry-out of the adder.

The bits in a CLA must pass through about $\log_2 n$ logic levels, compared with $2n$ for a ripple-carry adder. This is a substantial speed improvement, especially for a large n. Whereas the ripple-carry adder had n cells, however, the CLA has $2n$ cells, although in our layout they will take $n \log n$ space. The point is that a small investment in size pays off in a dramatic improvement in speed.

A number of technology-dependent modifications can improve CLAs. For example, if each node of the tree has three inputs instead of two, then the height

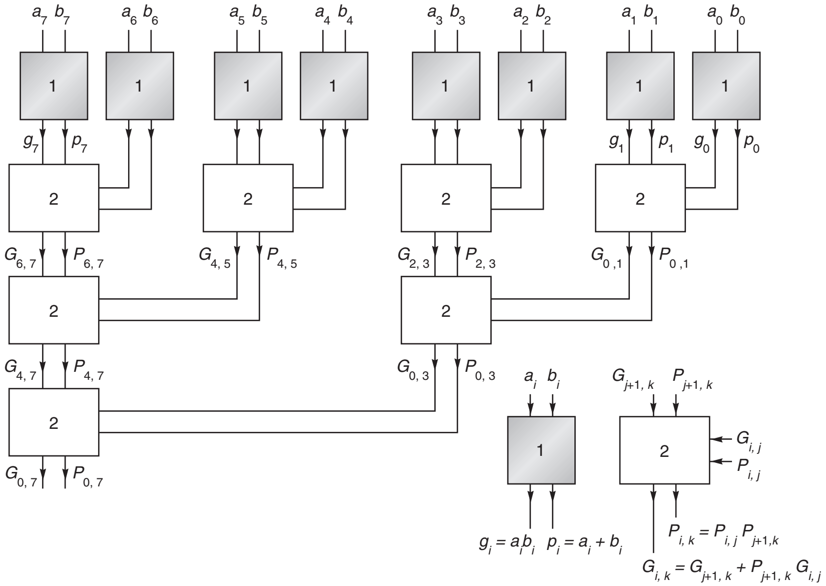

a_7 b_7
a_6 b_6
a_5 b_5
a_4 b_4
a_3 b_3
a_2 b_2
a_1 b_1
a_0 b_0
1
g_7 p_7
g_1 p_1 g_0 p_0
2
$G_{6,7}$ $P_{6,7}$
$G_{4,5}$ $P_{4,5}$
$G_{2,3}$ $P_{2,3}$
$G_{0,1}$ $P_{0,1}$
$G_{4,7}$ $P_{4,7}$
$G_{0,3}$ $P_{0,3}$
$G_{0,7}$ $P_{0,7}$
a_i b_i
$g_i = a_i b_i$ $p_i = a_i + b_i$
$G_{j+1,k}$ $P_{j+1,k}$
$G_{i,j}$
$P_{i,j}$
$P_{i,k} = P_{i,j} P_{j+1,k}$
$G_{i,k} = G_{j+1,k} + P_{j+1,k} G_{i,j}$

Figure J.15 First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of *P* and *G* are computed.

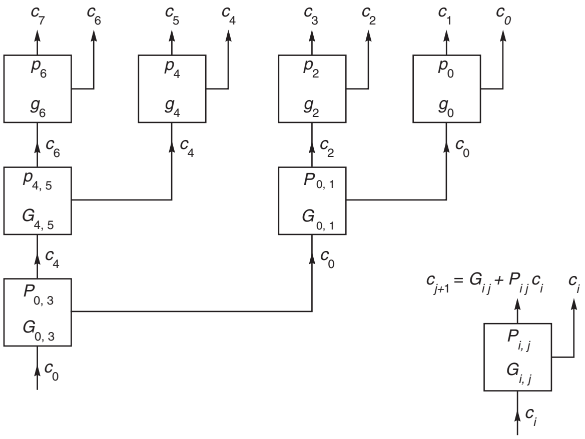

c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0
p_6 g_6
p_4 g_4
p_2 g_2
p_0 g_0
c_6 c_4 c_2 c_0
$P_{4,5}$ $G_{4,5}$
$P_{0,1}$ $G_{0,1}$
c_4 c_0
$P_{0,3}$ $G_{0,3}$
c_0
$c_{j+1} = G_{ij} + P_{ij} c_i$
c_i
$P_{i,j}$ $G_{i,j}$
c_i

Figure J.16 Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with *P* and *G* to form the carries.

of the tree will decrease from $\log_2 n$ to $\log_3 n$. Of course, the cells will be more complex and thus might operate more slowly, negating the advantage of the decreased height. For technologies where rippling works well, a hybrid design might be better. This is illustrated in Figure J.19. Carries ripple between adders

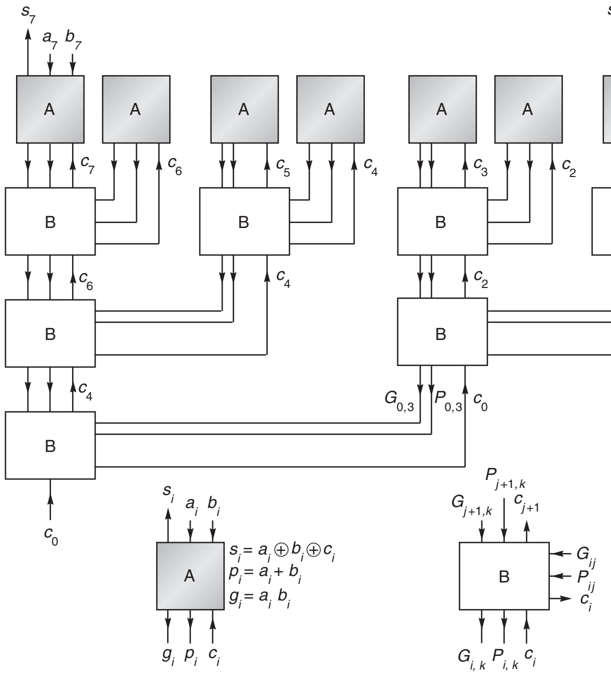

s7
a7 b7
s1
a1 b1
s0
a0 b0
A
c7
c6
c5
c4
c3
c2
c1
c0
B
c6
c4
c2
c0
B
c4
G0,3
P0,3
c0
B
c0
si
ai bi
A
si = ai ⊕ bi ⊕ ci
pi = ai + bi
gi = ai bi
gi pi ci
Pj+1,k
Gj+1,k
cj+1
B
Gij
Pij
ci
Gi,k Pi,k ci

Figure J.17 Complete carry-lookahead tree adder. This is the combination of Figures J.15 and J.16. The numbers to be added enter at the top, flow to the bottom to combine with c_0, and then flow back up to compute the sum bits.

at the top level, while the "B" boxes are the same as those in Figure J.17. This design will be faster if the time to ripple between four adders is faster than the time it takes to traverse a level of "B" boxes. (To make the pattern more clear, Figure J.19 shows a 16-bit adder, so the 8-bit adder of Figure J.17 corresponds to the right half of Figure J.19.)

Carry-Skip Adders

A *carry-skip adder* sits midway between a ripple-carry adder and a carry-lookahead adder, both in terms of speed and cost. (A carry-skip adder is not called a CSA, as that name is reserved for carry-save adders.) The motivation for this adder comes from examining the equations for P and G. For example,

$$P_{03} = p_0 p_1 p_2 p_3$$
$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Computing P is much simpler than computing G, and a carry-skip adder only computes the P's. Such an adder is illustrated in Figure J.18. Carries begin rippling

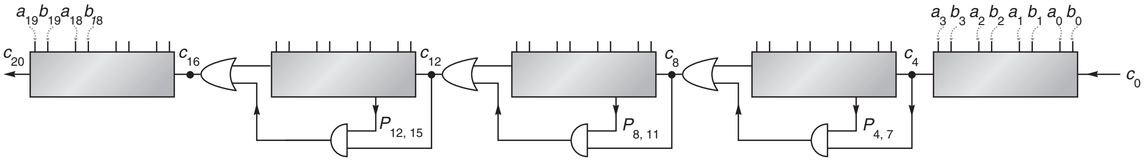


Figure J.18 Carry-skip adder. This is a 20-bit carry-skip adder ($n = 20$) with each block 4 bits wide ($k = 4$).

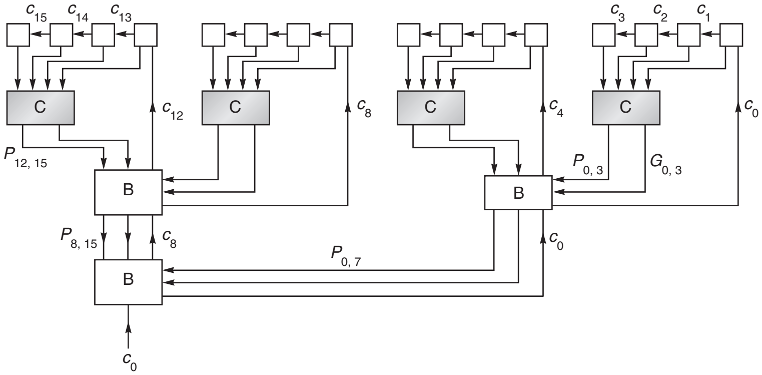


Figure J.19 Combination of CLA and ripple-carry adder. In the top row, carries ripple within each group of four boxes.

simultaneously through each block. If any block generates a carry, then the carry-out of a block will be true, even though the carry-in to the block may not be correct yet. If at the start of each add operation the carry-in to each block is 0, then no spurious carry-outs will be generated. Thus, the carry-out of each block can be thought of as if it were the G signal. Once the carry-out from the least-significant block is generated, it not only feeds into the next block but is also fed through the AND gate with the P signal from that next block. If the carry-out and P signals are both true, then the carry *skips* the second block and is ready to feed into the third block, and so on. The carry-skip adder is only practical if the carry-in signals can be easily cleared at the start of each operation—for example, by precharging in CMOS.

To analyze the speed of a carry-skip adder, let's assume that it takes 1 time unit for a signal to pass through two logic levels. Then it will take k time units for a carry to ripple across a block of size k, and it will take 1 time unit for a carry to skip a block. The longest signal path in the carry-skip adder starts with a carry being generated at the 0th position. If the adder is n bits wide, then it takes k time units to ripple through the first block, $n/k - 2$ time units to skip blocks, and k more to ripple through the last block. To be specific: if we have a 20-bit adder broken into groups of 4 bits, it will take $4 + (20/4 - 2) + 4 = 11$ time units to perform an add. Some experimentation reveals that there are more efficient ways to divide 20 bits into blocks. For example, consider five blocks with the least-significant 2 bits in the first block, the next 5 bits in the second block, followed by blocks of size 6, 5, and 2. Then the add time is reduced to 9 time units. This illustrates an important general principle. For a carry-skip adder, making the interior blocks larger will speed up the adder. In fact, the same idea of varying the block sizes can sometimes speed up other adder designs as well. Because of the large amount of rippling, a carry-skip adder is most appropriate for technologies where rippling is fast.

Carry-Select Adder

A *carry-select adder* works on the following principle: Two additions are performed in parallel, one assuming the carry-in is 0 and the other assuming the carry-in is 1. When the carry-in is finally known, the correct sum (which has been precomputed) is simply selected. An example of such a design is shown in Figure J.20. An 8-bit adder is divided into two halves, and the carry-out from the lower half is used to select the sum bits from the upper half. If each block is computing its sum using rippling (a linear time algorithm), then the design in Figure J.20 is twice as fast at 50% more cost. However, note that the c_4 signal must drive many muxes, which may be very slow in some technologies. Instead of dividing the adder into halves, it could be divided into quarters for a still further speedup. This is illustrated in Figure J.21. If it takes *k* time units for a block to add *k*-bit numbers, and if it takes 1 time unit to compute the mux input from the two carry-out signals, then for optimal operation each block should be 1 bit wider than the next, as shown in Figure J.21. Therefore, as in the carry-skip adder, the best design involves variable-size blocks.

As a summary of this section, the asymptotic time and space requirements for the different adders are given in Figure J.22. (The times for carry-skip and

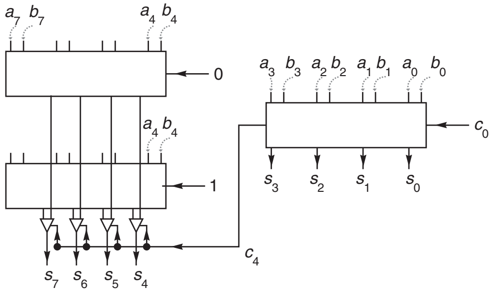


Figure J.20 Simple carry-select adder. At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that $c_4 = 0$ and once assuming $c_4 = 1$.

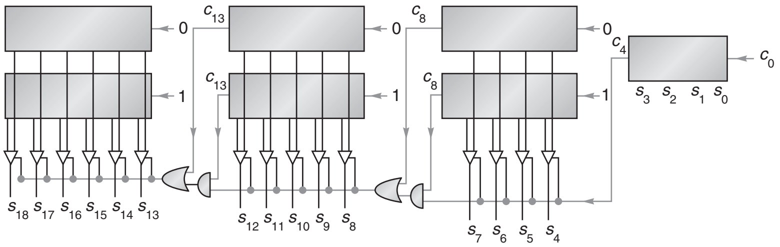

c_{13}
0
1
c_{13}
c_8
0
1
c_8
0
c_4
1
c_0
s_3 s_2 s_1 s_0
s_{18} s_{17} s_{16} s_{15} s_{14} s_{13}
s_{12} s_{11} s_{10} s_9 s_8
s_7 s_6 s_5 s_4

Figure J.21 Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

Adder	Time	Space
Ripple	$0(n)$	$0(n)$
CLA	$0(\log n)$	$0(n \log n)$
Carry-skip	$0(\sqrt{n})$	$0(n)$
Carry-select	$0(\sqrt{n})$	$0(n)$

Figure J.22 Asymptotic time and space requirements for four different types of adders.

carry-select come from a careful choice of block size. See Exercise J.26 for the carry-skip adder.) These different adders shouldn't be thought of as disjoint choices, but rather as building blocks to be used in constructing an adder. The utility of these different building blocks is highly dependent on the technology used. For example, the carry-select adder works well when a signal can drive many muxes, and the carry-skip adder is attractive in technologies where signals can be cleared at the start of each operation. Knowing the asymptotic behavior of adders is useful in understanding them, but relying too much on that behavior is a pitfall. The reason is that asymptotic behavior is only important as n grows very large. But n for an adder is the bits of precision, and double precision today is the same as it was 20 years ago—about 53 bits. Although it is true that as computers get faster, computations get longer—and thus have more rounding error, which in turn requires more precision—this effect grows very slowly with time.

J.9 Speeding Up Integer Multiplication and Division

The multiplication and division algorithms presented in Section J.2 are fairly slow, producing 1 bit per cycle (although that cycle might be a fraction of the CPU instruction cycle time). In this section, we discuss various techniques for higher-performance multiplication and division, including the division algorithm used in the Pentium chip.

Shifting over Zeros

Although the technique of shifting over zeros is not currently used much, it is instructive to consider. It is distinguished by the fact that its execution time is operand dependent. Its lack of use is primarily attributable to its failure to offer enough speedup over bit-at-a-time algorithms. In addition, pipelining, synchronization with the CPU, and good compiler optimization are difficult with algorithms that run in variable time. In multiplication, the idea behind shifting over zeros is to add logic that detects when the low-order bit of the A register is 0 (see Figure J.2(a) on page J-4) and, if so, skips the addition step and proceeds directly to the shift step—hence the term *shifting over zeros*.

What about shifting for division? In nonrestoring division, an ALU operation (either an addition or subtraction) is performed at every step. There appears to be no opportunity for skipping an operation. But think about division this way: To compute a/b, subtract multiples of b from a, and then report how many subtractions were done. At each stage of the subtraction process the remainder must fit into the P register of Figure J.2(b) (page J-4). In the case when the remainder is a small positive number, you normally subtract b; but suppose instead you only shifted the remainder and subtracted b the next time. As long as the remainder was sufficiently small (its high-order bit 0), after shifting it still would fit into the P register, and no information would be lost. However, this method does require changing the way we keep track of the number of times b has been subtracted from a. This idea usually goes under the name of *SRT division*, for Sweeney, Robertson, and Tocher, who independently proposed algorithms of this nature. The main extra complication of SRT division is that the quotient bits cannot be determined immediately from the sign of P at each step, as they can be in ordinary nonrestoring division.

More precisely, to divide a by b where a and b are n-bit numbers, load a and b into the A and B registers, respectively, of Figure J.2 (page J-4).

SRT Division

1. If B has k leading zeros when expressed using n bits, shift all the registers left k bits.
2. For $i=0,\ n-1$,
    a) If the top three bits of P are equal, set $q_i=0$ and shift (P,A) one bit left.
    b) If the top three bits of P are not all equal and P is negative, set $q_i=-1$ (also written as $\overline{1}$), shift (P,A) one bit left, and add B.
    c) Otherwise set $q_i=1$, shift (P,A) one bit left, and subtract B.

    End loop
3. If the final remainder is negative, correct the remainder by adding B, and correct the quotient by subtracting 1 from q_0. Finally, the remainder must be shifted k bits right, where k is the initial shift.

P	A	
00000	1000	Divide $8 = 1000$ by $3 = 0011$. B contains 0011.
00010	0000	Step 1: B had two leading 0 s, so shift left by 2. B now contains 1100.
		Step 2.1: Top three bits are equal. This is case (a), so
00100	000**0**	set $q_0 = 0$ and shift.
		Step 2.2: Top three bits not equal and $P \geq 0$ is case (c), so
01000	00**01**	set $q_1 = 1$ and shift.
+ 10100		Subtract B.
11100	00**01**	Step 2.3: Top bits equal is case (a), so
11000	0**010**	set $q_2 = 0$ and shift.
		Step 2.4: Top three bits unequal is case (b), so
10000	$\mathbf{010\bar{1}}$	set $q_3 = -1$ and shift.
+ 01100		Add B.
11100		Step 3. remainder is negative so restore it and subtract 1 from q.
+ 01100		
01000		Must undo the shift in step 1, so right-shift by 2 to get true remainder. Remainder $= 10$, quotient $= 010\bar{1} - 1 = 0010$.

Figure J.23 SRT division of $1000_2/0011_2$. The quotient bits are shown in bold, using the notation 1 for −1.

A numerical example is given in Figure J.23. Although we are discussing integer division, it helps in explaining the algorithm to imagine the binary point just left of the most-significant bit. This changes Figure J.23 from $01000_2/0011_2$ to $0.1000_2/.0011_2$. Since the binary point is changed in both the numerator and denominator, the quotient is not affected. The (P,A) register pair holds the remainder and is a two's complement number. For example, if P contains 11110_2 and $A = 0$, then the remainder is $1.1110_2 = -1/8$. If r is the value of the remainder, then $-1 \leq r < 1$.

Given these preliminaries, we can now analyze the SRT division algorithm. The first step of the algorithm shifts b so that $b \geq 1/2$. The rule for which ALU operation to perform is this: If $-1/4 \leq r < 1/4$ (true whenever the top three bits of P are equal), then compute $2r$ by shifting (P,A) left one bit; if $r < 0$ (and hence $r < -1/4$, since otherwise it would have been eliminated by the first condition), then compute $2r + b$ by shifting and then adding; if $r \geq 1/4$ and subtract b from $2r$. Using $b \geq 1/2$, it is easy to check that these rules keep $-1/2 \leq r < 1/2$. For nonrestoring division, we only have $|r| \leq b$, and we need P to be $n+1$ bits wide. But, for SRT division, the bound on r is tighter, namely, $-1/2 \leq r < 1/2$. Thus, we can save a bit by eliminating the high-order bit of P (and b and the adder). In particular, the test for equality of the top three bits of P becomes a test on just two bits.

The algorithm might change slightly in an implementation of SRT division. After each ALU operation, the P register can be shifted as many places as necessary to make either $r \geq 1/4$ or $r < -1/4$. By shifting k places, k quotient bits are set equal to zero all at once. For this reason SRT division is sometimes described as one that keeps the remainder normalized to $|r| \geq 1/4$.

Notice that the value of the quotient bit computed in a given step is based on which operation is performed in that step (which in turn depends on the result of the operation from the previous step). This is in contrast to nonrestoring division, where the quotient bit computed in the *i*th step depends on the result of the operation in the same step. This difference is reflected in the fact that when the final remainder is negative, the last quotient bit must be adjusted in SRT division, but not in nonrestoring division. However, the key fact about the quotient bits in SRT division is that they can include $\overline{1}$. Although Figure J.23 shows the quotient bits being stored in the low-order bits of A, an actual implementation can't do this because you can't fit the three values -1, 0, 1 into one bit. Furthermore, the quotient must be converted to ordinary two's complement in a full adder. A common way to do this is to accumulate the positive quotient bits in one register and the negative quotient bits in another, and then subtract the two registers after all the bits are known. Because there is more than one way to write a number in terms of the digits -1, 0, 1, SRT division is said to use a *redundant* quotient representation.

The differences between SRT division and ordinary nonrestoring division can be summarized as follows:

1. ALU decision rule—In nonrestoring division, it is determined by the sign of P; in SRT, it is determined by the two most-significant bits of P.
2. Final quotient—In nonrestoring division, it is immediate from the successive signs of P; in SRT, there are three quotient digits (1, 0, $\overline{1}$), and the final quotient must be computed in a full *n*-bit adder.
3. Speed—SRT division will be faster on operands that produce zero quotient bits.

The simple version of the SRT division algorithm given above does not offer enough of a speedup to be practical in most cases. However, later on in this section we will study variants of SRT division that are quite practical.

Speeding Up Multiplication with a Single Adder

As mentioned before, shifting-over-zero techniques are not used much in current hardware. We now discuss some methods that are in widespread use. Methods that increase the speed of multiplication can be divided into two classes: those that use a single adder and those that use multiple adders. Let's first discuss techniques that use a single adder.

In the discussion of addition we noted that, because of carry propagation, it is not practical to perform addition with two levels of logic. Using the cells of Figure J.17, adding two 64-bit numbers will require a trip through seven cells to compute the *P*'s and *G*'s and seven more to compute the carry bits, which will require at least 28 logic levels. In the simple multiplier of Figure J.2 on page J-4, each multiplication step passes through this adder. The amount of computation in each step can be dramatically reduced by using *carry-save adders* (CSAs). A carry-save adder is simply a collection of *n* independent full adders. A multiplier using

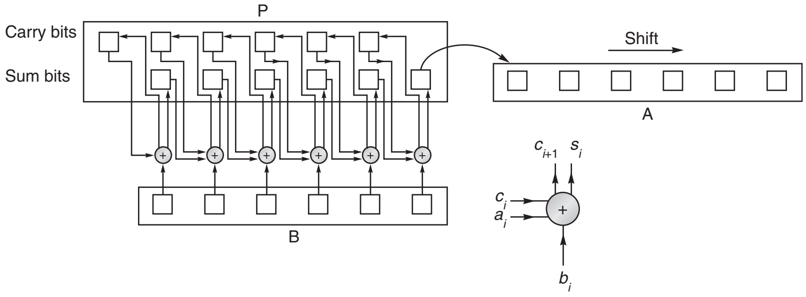


Figure J.24 Carry-save multiplier. Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

such an adder is illustrated in Figure J.24. Each circle marked "+" is a single-bit full adder, and each box represents one bit of a register. Each addition operation results in a pair of bits, stored in the sum and carry parts of P. Since each add is independent, only two logic levels are involved in the add—a vast improvement over 28.

To operate the multiplier in Figure J.24, load the sum and carry bits of P with zero and perform the first ALU operation. (If Booth recoding is used, it might be a subtraction rather than an addition.) Then shift the low-order sum bit of P into A, as well as shifting A itself. The $n-1$ high-order bits of P don't need to be shifted because on the next cycle the sum bits are fed into the next lower-order adder. Each addition step is substantially increased in speed, since each add cell is working independently of the others, and no carry is propagated.

There are two drawbacks to carry-save adders. First, they require more hardware because there must be a copy of register P to hold the carry outputs of the adder. Second, after the last step, the high-order word of the result must be fed into an ordinary adder to combine the sum and carry parts. One way to accomplish this is by feeding the output of P into the adder used to perform the addition operation. Multiplying with a carry-save adder is sometimes called *redundant multiplication* because P is represented using two registers. Since there are many ways to represent P as the sum of two registers, this representation is redundant. The term *carry-propagate adder* (CPA) is used to denote an adder that is not a CSA. A propagate adder may propagate its carries using ripples, carry-lookahead, or some other method.

Another way to speed up multiplication without using extra adders is to examine k low-order bits of A at each step, rather than just one bit. This is often called *higher-radix multiplication*. As an example, suppose that $k=2$. If the pair of bits is 00, add 0 to P; if it is 01, add B. If it is 10, simply shift b one bit left before adding it to P. Unfortunately, if the pair is 11, it appears we would have to compute $b+2b$. But this can be avoided by using a higher-radix version of Booth recoding. Imagine A as a base 4 number: When the digit 3 appears, change it to $\overline{1}$ and add 1 to the next higher digit to compensate. An extra benefit of using this scheme is that just like ordinary Booth recoding, it works for negative as well as positive integers (Section J.2).

The precise rules for radix-4 Booth recoding are given in Figure J.25. At the ith multiply step, the two low-order bits of the A register contain a_{2i} and a_{2i+1}. These two bits, together with the bit just shifted out (a_{2i-1}), are used to select the multiple of b that must be added to the P register. A numerical example is given in Figure J.26. Another name for this multiplication technique is *overlapping triplets,* since it looks at 3 bits to determine what multiple of b to use, whereas ordinary Booth recoding looks at 2 bits.

Besides having more complex control logic, overlapping triplets also requires that the P register be 1 bit wider to accommodate the possibility of $2b$ or $-2b$ being added to it. It is possible to use a radix-8 (or even higher) version of Booth recoding. In that case, however, it would be necessary to use the multiple 3B as a potential summand. Radix-8 multipliers normally compute 3B once and for all at the beginning of a multiplication operation.

Low-order bits of A		**Last bit shifted out**	
$2i+1$	$2i$	$2i-1$	Multiple
0	0	0	0
0	0	1	$+b$
0	1	0	$+b$
0	1	1	$+2b$
1	0	0	$-2b$
1	0	1	$-b$
1	1	0	$-b$
1	1	1	0

Figure J.25 Multiples of *b* to use for radix-4 Booth recoding. For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is $-b$, obtained from the second-to-last row of the table.

P	**A**	**L**	
00000	1001		Multiply −7 = 1001 times −5 = 1011. B contains 1011.
+ 11011			Low-order bits of A are 0, 1; L = 0, so add B.
11011	1001		
11110	1110	0	Shift right by two bits, shifting in 1 s on the left.
+ 01010			Low-order bits of A are 1, 0; L = 0, so add −2b.
01000	1110	0	
00010	0011	1	Shift right by two bits.
			Product is 35 = 0100011.

Figure J.26 Multiplication of −7 times −5 using radix-4 Booth recoding. The column labeled L contains the last bit shifted out the right end of A.

Faster Multiplication with Many Adders

If the space for many adders is available, then multiplication speed can be improved. Figure J.27 shows a simple array multiplier for multiplying two 5-bit numbers, using three CSAs and one propagate adder. Part (a) is a block diagram of the kind we will use throughout this section. Parts (b) and (c) show the adder in more detail. All the inputs to the adder are shown in (b); the actual adders with their interconnections are shown in (c). Each row of adders in (c) corresponds to a box in

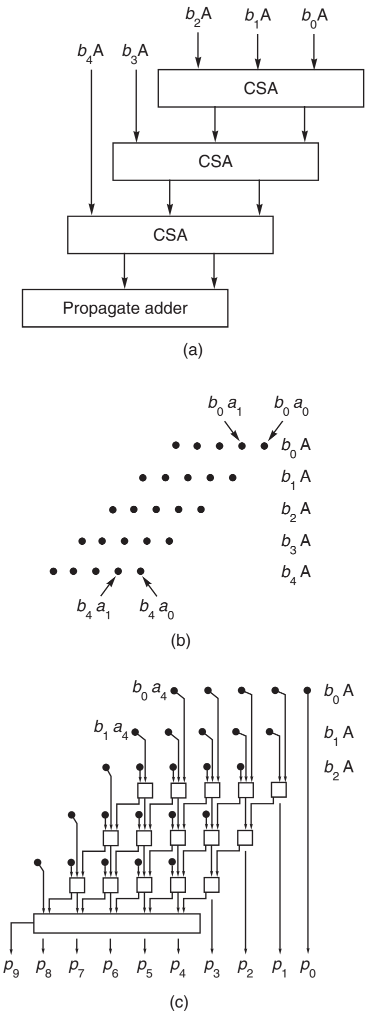

b_2A
b_1A
b_0A
b_4A
b_3A
CSA
CSA
CSA
Propagate adder
(a)
$b_0 a_1$
$b_0 a_0$
b_0 A
b_1 A
b_2 A
b_3 A
b_4 A
$b_4 a_1$
$b_4 a_0$
(b)
$b_0 a_4$
$b_1 a_4$
b_0 A
b_1 A
b_2 A
p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0
(c)

Figure J.27 An array multiplier. The 5-bit number in A is multiplied by $b_4b_3b_2b_1b_0$. Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

(a). The picture is "twisted" so that bits of the same significance are in the same column. In an actual implementation, the array would most likely be laid out as a square instead.

The array multiplier in Figure J.27 performs the same number of additions as the design in Figure J.24, so its latency is not dramatically different from that of a single carry-save adder. However, with the hardware in Figure J.27, multiplication can be pipelined, increasing the total throughput. On the other hand, although this level of pipelining is sometimes used in array processors, it is not used in any of the single-chip, floating-point accelerators discussed in Section J.10. Pipelining is discussed in general in Appendix C and by Kogge [1981] in the context of multipliers.

Sometimes the space budgeted on a chip for arithmetic may not hold an array large enough to multiply two double-precision numbers. In this case, a popular design is to use a two-pass arrangement such as the one shown in Figure J.28. The first pass through the array "retires" 5 bits of B. Then the result of this first pass is fed back into the top to be combined with the next three summands. The result of this second pass is then fed into a CPA. This design, however, loses the ability to be pipelined.

If arrays require as many addition steps as the much cheaper arrangements in Figures J.2 and J.24, why are they so popular? First of all, using an array has a smaller latency than using a single adder—because the array is a combinational circuit, the signals flow through it directly without being clocked. Although the two-pass adder of Figure J.28 would normally still use a clock, the cycle time for passing through *k* arrays can be less than *k* times the clock that would be needed for designs like the ones in Figures J.2 or J.24. Second, the array is amenable to various schemes for further speedup. One of them is shown in Figure J.29. The idea of this design is that two adds proceed in parallel or, to put it another way, each stream passes through only half the adders. Thus, it runs at almost twice

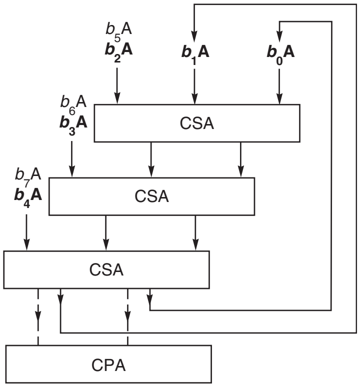


Figure J.28 Multipass array multiplier. Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure J.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.

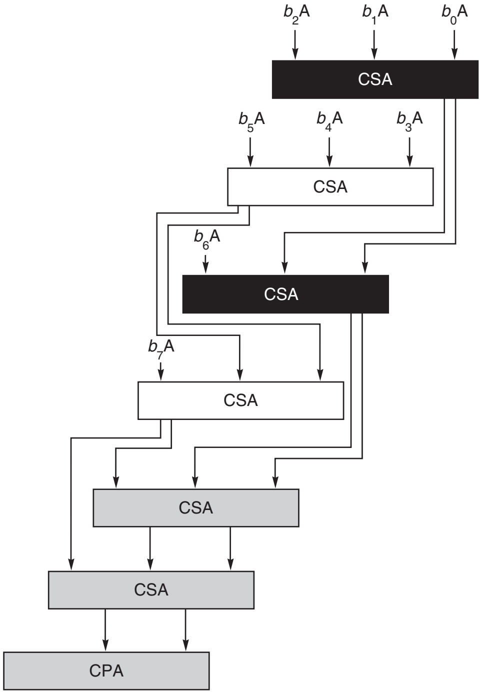


Figure J.29 Even/odd array. The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

the speed of the multiplier in Figure J.27. This *even/odd* multiplier is popular in VLSI because of its regular structure. Arrays can also be speeded up using asynchronous logic. One of the reasons why the multiplier of Figure J.2 (page J-4) needs a clock is to keep the output of the adder from feeding back into the input of the adder before the output has fully stabilized. Thus, if the array in Figure J.28 is long enough so that no signal can propagate from the top through the bottom in the time it takes for the first adder to stabilize, it may be possible to avoid clocks altogether. Williams et al. [1987] discussed a design using this idea, although it is for dividers instead of multipliers.

The techniques of the previous paragraph still have a multiply time of 0(n), but the time can be reduced to log n using a tree. The simplest tree would combine pairs of summands $b_0\text{A}\cdots b_{n-1}\text{A}$, cutting the number of summands from n to $n/2$. Then these $n/2$ numbers would be added in pairs again, reducing to $n/4$, and so on, and resulting in a single sum after log n steps. However, this simple binary tree idea doesn't map into full (3,2) adders, which reduce three inputs to two rather than reducing two inputs to one. A tree that does use full adders, known as a *Wallace tree*, is shown in Figure J.30. When computer arithmetic units were built out of

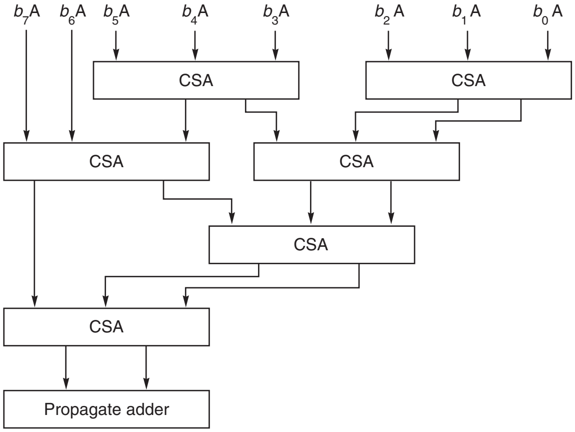


Figure J.30 Wallace tree multiplier. An example of a multiply tree that computes a product in 0(log *n*) steps.

MSI parts, a Wallace tree was the design of choice for high-speed multipliers. There is, however, a problem with implementing it in VLSI. If you try to fill in all the adders and paths for the Wallace tree of Figure J.30, you will discover that it does not have the nice, regular structure of Figure J.27. This is why VLSI designers have often chosen to use other log n designs such as the *binary tree multiplier*, which is discussed next.

The problem with adding summands in a binary tree is coming up with a (2,1) adder that combines two digits and produces a single-sum digit. Because of carries, this isn't possible using binary notation, but it can be done with some other representation. We will use the *signed-digit representation* 1, $\overline{1}$, and 0, which we used previously to understand Booth's algorithm. This representation has two costs. First, it takes 2 bits to represent each signed digit. Second, the algorithm for adding two signed-digit numbers a_i and b_i is complex and requires examining $a_i a_{i-1} a_{i-2}$ and $b_i b_{i-1} b_{i-2}$. Although this means you must look 2 bits back, in binary addition you might have to look an arbitrary number of bits back because of carries.

We can describe the algorithm for adding two signed-digit numbers as follows. First, compute sum and carry bits s_i and c_{i+1} using Figure J.31. Then compute the final sum as $s_i + c_i$. The tables are set up so that this final sum does not generate a carry.

Example What is the sum of the signed-digit numbers $1\overline{1}0_2$ and 001_2?

Answer The two low-order bits sum to $0 + 1 = 1\overline{1}$, the next pair sums to $\overline{1} + 0 = 0\overline{1}$, and the high-order pair sums to $1 + 0 = 01$, so the sum is $1\overline{1} + 0\overline{1}0 + 0100 = 10\overline{1}_2$.

$$\begin{array}{r} 1 \\ +\,1 \\ \hline 1\,0 \end{array} \qquad \begin{array}{r} 1 \\ +\,\bar{1} \\ \hline 0\,0 \end{array} \qquad \begin{array}{r} \bar{1} \\ +\,\bar{1} \\ \hline \bar{1}\,0 \end{array} \qquad \begin{array}{r} 0 \\ +\,0 \\ \hline 0\,0 \end{array} \qquad \begin{array}{rcc} & \bar{1} & x \\ + & 0 & y \\ \hline 1 & \bar{1} & \\ 0 & 1 & \end{array} \quad \begin{array}{l} \text{if } x \geq 0 \text{ and} \\ y \geq 0 \text{ otherwise} \end{array} \qquad \begin{array}{rcc} & \bar{1} & x \\ + & 0 & y \\ \hline 1 & \bar{1} & \\ \bar{1} & 1 & \end{array} \quad \begin{array}{l} \text{if } x \geq 0 \text{ and} \\ y \geq 0 \text{ otherwise} \end{array}$$

Figure J.31 Signed-digit addition table. The leftmost sum shows that when computing $1+1$, the sum bit is 0 and the carry bit is 1.

This, then, defines a (2,1) adder. With this in hand, we can use a straightforward binary tree to perform multiplication. In the first step it adds $b_0\mathrm{A}+b_1\mathrm{A}$ in parallel with $b_2\mathrm{A}+b_3\mathrm{A}, \ldots, b_{n-2}\mathrm{A}+b_{n-1}\mathrm{A}$. The next step adds the results of these sums in pairs, and so on. Although the final sum must be run through a carry-propagate adder to convert it from signed-digit form to two's complement, this final add step is necessary in any multiplier using CSAs.

To summarize, both Wallace trees and signed-digit trees are log n multipliers. The Wallace tree uses fewer gates but is harder to lay out. The signed-digit tree has a more regular structure, but requires 2 bits to represent each digit and has more complicated add logic. As with adders, it is possible to combine different multiply techniques. For example, Booth recoding and arrays can be combined. In Figure J.27 instead of having each input be $b_i\mathrm{A}$, we could have it be $b_ib_{i-1}\mathrm{A}$. To avoid having to compute the multiple $3b$, we can use Booth recoding.

Faster Division with One Adder

The two techniques we discussed for speeding up multiplication with a single adder were carry-save adders and higher-radix multiplication. However, there is a difficulty when trying to utilize these approaches to speed up nonrestoring division. If the adder in Figure J.2(b) on page J-4 is replaced with a carry-save adder, then P will be replaced with two registers, one for the sum bits and one for the carry bits (compare with the multiplier in Figure J.24). At the end of each cycle, the sign of P is uncertain (since P is the unevaluated sum of the two registers), yet it is the sign of P that is used to compute the quotient digit and decide the next ALU operation. When a higher radix is used, the problem is deciding what value to subtract from P. In the paper-and-pencil method, you have to guess the quotient digit. In binary division, there are only two possibilities. We were able to finesse the problem by initially guessing one and then adjusting the guess based on the sign of P. This doesn't work in higher radices because there are more than two possible quotient digits, rendering quotient selection potentially quite complicated: You would have to compute all the multiples of b and compare them to P.

Both the carry-save technique and higher-radix division can be made to work if we use a redundant quotient representation. Recall from our discussion of SRT division (page J-45) that by allowing the quotient digits to be -1, 0, or 1, there is often a choice of which one to pick. The idea in the previous algorithm was to choose 0 whenever possible, because that meant an ALU operation could be

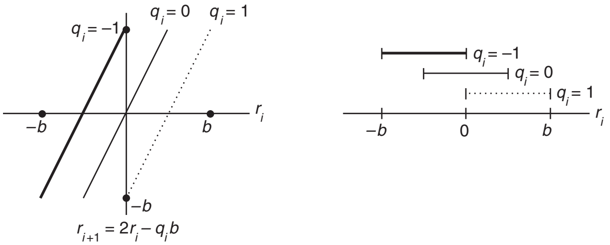


Figure J.32 Quotient selection for radix-2 division. The *x*-axis represents the *i*th remainder, which is the quantity in the (P,A) register pair. The *y*-axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of r_i values for which it is permissible to select the associated value of q_i.

skipped. In carry-save division, the idea is that, because the remainder (which is the value of the (P,A) register pair) is not known exactly (being stored in carry-save form), the exact quotient digit is also not known. But, thanks to the redundant representation, the remainder doesn't have to be known precisely in order to pick a quotient digit. This is illustrated in Figure J.32, where the *x*-axis represents r_i, the remainder after *i* steps. The line labeled $q_i = 1$ shows the value that r_{i+1} would be if we chose $q_i = 1$, and similarly for the lines $q_i = 0$ and $q_i = -1$. We can choose any value for q_i, as long as $r_{i+1} = 2r_i - q_i b$ satisfies $|r_{i+1}| \leq b$. The allowable ranges are shown in the right half of Figure J.32. This shows that you don't need to know the precise value of r_i in order to choose a quotient digit q_i. You only need to know that r lies in an interval small enough to fit entirely within one of the overlapping bars shown in the right half of Figure J.32.

This is the basis for using carry-save adders. Look at the high-order bits of the carry-save adder and sum them in a propagate adder. Then use this approximation of r (together with the divisor, b) to compute q_i, usually by means of a lookup table. The same technique works for higher-radix division (whether or not a carry-save adder is used). The high-order bits P can be used to index a table that gives one of the allowable quotient digits.

The design challenge when building a high-speed SRT divider is figuring out how many bits of P and B need to be examined. For example, suppose that we take a radix of 4, use quotient digits of 2, 1, 0, $\overline{1}$, $\overline{2}$, but have a propagate adder. How many bits of P and B need to be examined? Deciding this involves two steps. For ordinary radix-2 nonrestoring division, because at each stage $|r| \leq b$, the P buffer won't overflow. But, for radix 4, $r_{i+1} = 4r_i - q_i b$ is computed at each stage, and if r_i is near b, then $4r_i$ will be near $4b$, and even the largest quotient digit will not bring r back to the range $|r_{i+1}| \leq b$. In other words, the remainder might grow without bound. However, restricting $|r_i| \leq 2b/3$ makes it easy to check that r_i will stay bounded.

After figuring out the bound that r_i must satisfy, we can draw the diagram in Figure J.33, which is analogous to Figure J.32. For example, the diagram shows

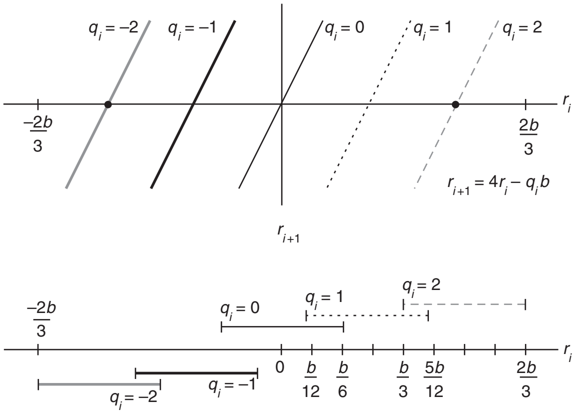


Figure J.33 Quotient selection for radix-4 division with quotient digits −2, −1, 0, 1, 2.

that if r_i is between (1/12)*b* and (5/12)*b*, we can pick $q = 1$, and so on. Or, to put it another way, if *r*/*b* is between 1/12 and 5/12, we can pick $q = 1$. Suppose the divider examines 5 bits of P (including the sign bit) and 4 bits of *b* (ignoring the sign, since it is always nonnegative). The interesting case is when the high bits of P are 00011*xxx*⋯, while the high bits of *b* are 1001*xxx*⋯. Imagine the binary point at the left end of each register. Since we truncated, *r* (the value of P concatenated with A) could have a value from 0.0011_2 to 0.0100_2, and *b* could have a value from $.1001_2$ to $.1010_2$. Thus, *r*/*b* could be as small as $0.0011_2/.1010_2$ or as large as $0.0100_2/.1001_2$, but $0.0011_2/.1010_2 = 3/10 < 1/3$ would require a quotient bit of 1, while $0.0100_2/.1001_2 = 4/9 > 5/12$ would require a quotient bit of 2. In other words, 5 bits of P and 4 bits of *b* aren't enough to pick a quotient bit. It turns out that 6 bits of P and 4 bits of *b* are enough. This can be verified by writing a simple program that checks all the cases. The output of such a program is shown in Figure J.34.

Example Using 8-bit registers, compute 149/5 using radix-4 SRT division.

Answer Follow the SRT algorithm on page J-45, but replace the quotient selection rule in step 2 with one that uses Figure J.34. See Figure J.35.

The Pentium uses a radix-4 SRT division algorithm like the one just presented, except that it uses a carry-save adder. Exercises J.34(c) and J.35 explore this in detail. Although these are simple cases, all SRT analyses proceed in the same way. First compute the range of r_i, then plot r_i against r_{i+1} to find the quotient ranges, and finally write a program to compute how many bits are necessary. (It is sometimes also possible to compute the required number of bits analytically.) Various details need to be considered in building a practical SRT divider.

b	Range of P		*q*	*b*	Range of P		*q*
8	−12	−7	−2	12	−18	−10	−2
8	−6	−3	−1	12	−10	−4	−1
8	−2	1	0	12	−4	3	0
8	2	5	1	12	3	9	1
8	6	11	2	12	9	17	2
9	−14	−8	−2	13	−19	−11	−2
9	−7	−3	−1	13	−10	−4	−1
9	−3	2	0	13	−4	3	0
9	2	6	1	13	3	9	1
9	7	13	2	13	10	18	2
10	−15	−9	−2	14	−20	−11	−2
10	−8	−3	−1	14	−11	−4	−1
10	−3	2	0	14	−4	3	0
10	2	7	1	14	3	10	1
10	8	14	2	14	10	19	2
11	−16	−9	−2	15	−22	−12	−2
11	−9	−3	−1	15	−12	−4	−1
11	−3	2	0	15	−5	4	0
11	2	8	1	15	3	11	1
11	8	15	2	15	11	21	2

Figure J.34 Quotient digits for radix-4 SRT division with a propagate adder. The top row says that if the high-order 4 bits of *b* are $1000_2 = 8$, and if the top 6 bits of P are between $110100_2 = -12$ and $111001_2 = -7$, then −2 is a valid quotient digit.

For example, the quotient lookup table has a fairly regular structure, which means it is usually cheaper to encode it as a PLA rather than in ROM. For more details about SRT division, see Burgess and Williams [1995].

J.10 Putting It All Together

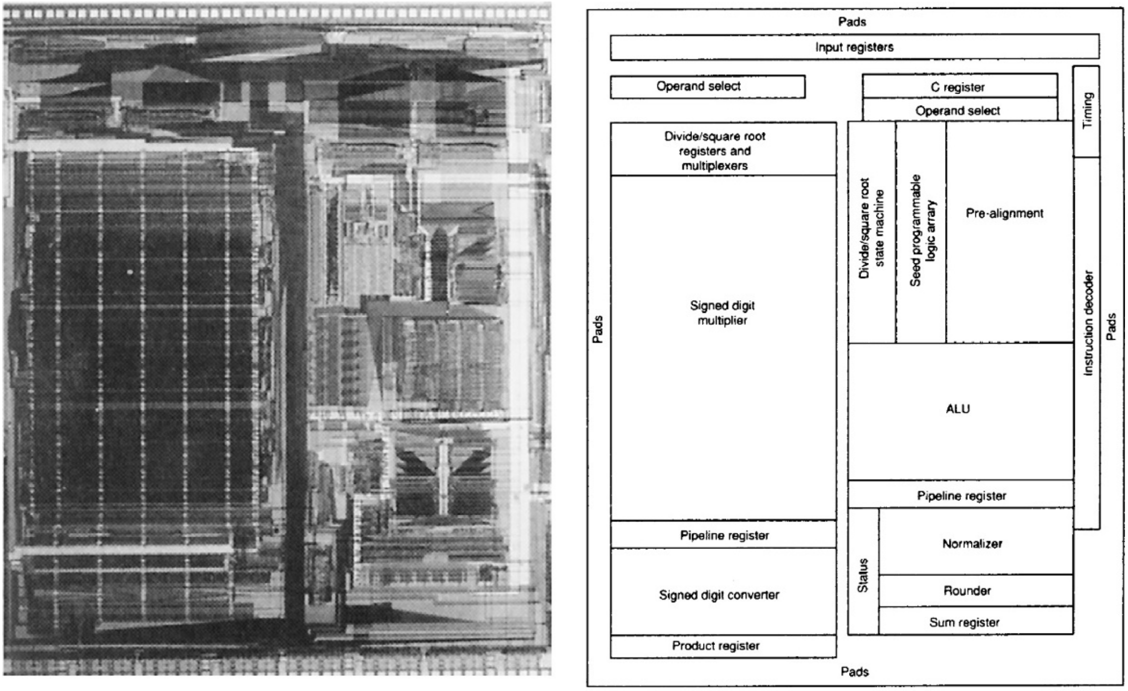
In this section, we will compare the Weitek 3364, the MIPS R3010, and the Texas Instruments 8847 (see Figures J.36 and J.37). In many ways, these are ideal chips to compare. They each implement the IEEE standard for addition, subtraction,

P	A		
000000000	10010101	Divide 149 by 5. B contains 00000101.	
000010010	10100000	Step 1:	B had 5 leading 0s, so shift left by 5. B now contains 10100000, so use $b=10$ section of table.
		Step 2.1:	Top 6 bits of P are 2, so shift left by 2. From table, can pick q to be 0 or 1. Choose $q_0=0$.
001001010	1000000**0**		
		Step 2.2:	Top 6 bits of P are 9, so shift left 2. $q_1=2$.
100101010	0000**02**		
+ 011000000			Subtract 2*b*.
111101010	0000**02**	Step 2.3:	Top bits $=-3$, so shift left 2. Can pick 0 or -1 for q, pick $q_2=0$.
110101000	00**020**		
		Step 2.4:	Top bits $=-11$, so shift left 2. $q_3=-2$.
010100000	**0202**		
+ 101000000			Add 2*b*.
111100000		Step 3:	Remainder is negative, so restore by adding b and subtract 1 from q.
+ 010100000			
010000000		Answer:	$q=020\overline{2}-1=29$
			To get remainder, undo shift in step 1 so remainder $=010000000 >> 5 = 4$.

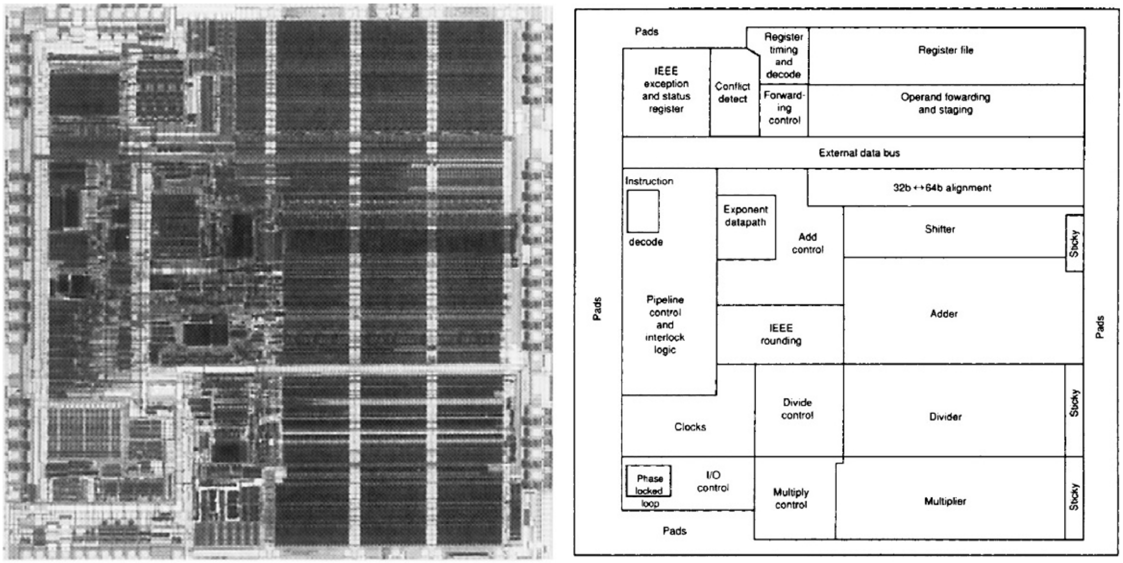
Figure J.35 Example of radix-4 SRT division. Division of 149 by 5.

Features	MIPS R3010	Weitek 3364	TI 8847
Clock cycle time (ns)	40	50	30
Size (mil^2)	114,857	147,600	156,180
Transistors	75,000	165,000	180,000
Pins	84	168	207
Power (watts)	3.5	1.5	1.5
Cycles/add	2	2	2
Cycles/mult	5	2	3
Cycles/divide	19	17	11
Cycles/square root	–	30	14

Figure J.36 Summary of the three floating-point chips discussed in this section. The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.


Pads
Input registers
Operand select
C register
Operand select
Timing
Divide/square root registers and multiplexers
Divide/square root state machine
Seed programmable logic array
Pre-alignment
Signed digit multiplier
Pads
Instruction decoder
Pads
ALU
Pipeline register
Pipeline register
Normalizer
Status
Signed digit converter
Rounder
Sum register
Product register
Pads

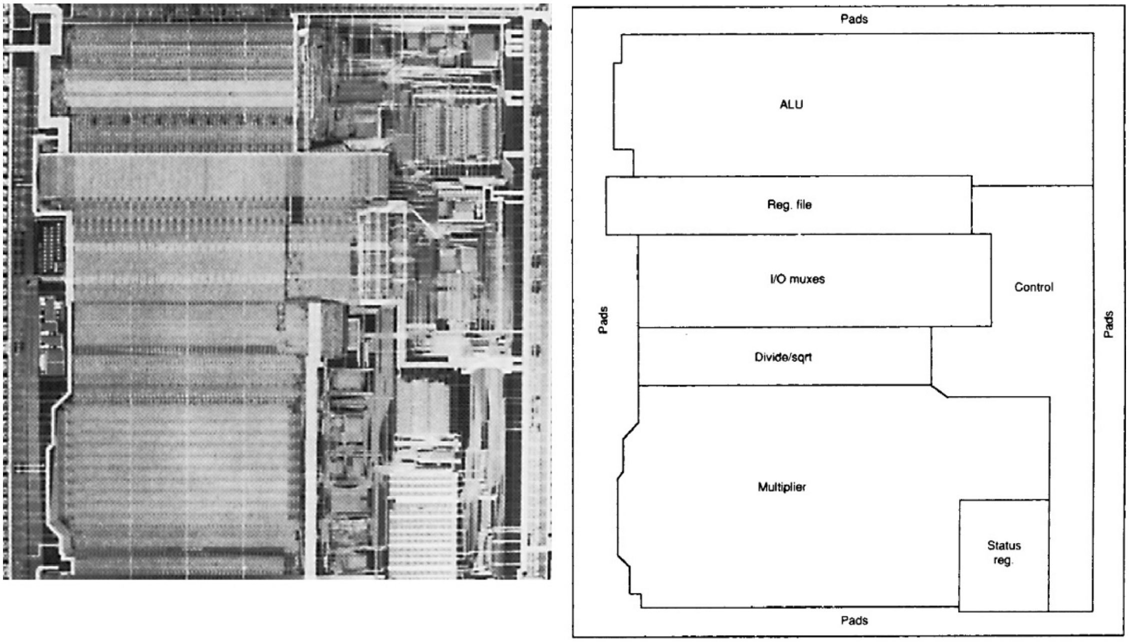
TI 8847


Pads
Register timing and decode
Register file
IEEE exception and status register
Conflict detect
Forward-ing control
Operand fowarding and staging
External data bus
Instruction
decode
32b ↔ 64b alignment
Exponent datapath
Add control
Shifter
Sticky
Pads
Pipeline control and interlock logic
IEEE rounding
Adder
Pads
Divide control
Clocks
Divider
Sticky
Phase locked loop
I/O control
Multiply control
Multiplier
Sticky
Pads

MIPS R3010

Figure J.37 Chip layout for the TI 8847, MIPS R3010, and Weitek 3364. In the left-hand columns are the photomicrographs; the right-hand columns show the corresponding floor plans.

(Continued)



Weitek 3364

Figure J.37 *(Continued)*

multiplication, and division on a single chip. All were introduced in 1988 and run with a cycle time of about 40 nanoseconds. However, as we will see, they use quite different algorithms. The Weitek chip is well described in Birman et al. [1990], the MIPS chip is described in less detail in Rowen, Johnson, and Ries [1988], and details of the TI chip can be found in Darley et al. [1989].

These three chips have a number of things in common. They perform addition and multiplication in parallel, and they implement neither extended precision nor a remainder step operation. (Recall from Section J.6 that it is easy to implement the IEEE remainder function in software if a remainder step instruction is available.) The designers of these chips probably decided not to provide extended precision because the most influential users are those who run portable codes, which can't rely on extended precision. However, as we have seen, extended precision can make for faster and simpler math libraries.

In the summary of the three chips given in Figure J.36, note that a higher transistor count generally leads to smaller cycle counts. Comparing the cycles/op numbers needs to be done carefully, because the figures for the MIPS chip are those for a complete system (R3000/3010 pair), while the Weitek and TI numbers are for stand-alone chips and are usually larger when used in a complete system.

The MIPS chip has the fewest transistors of the three. This is reflected in the fact that it is the only chip of the three that does not have any pipelining or hardware square root. Further, the multiplication and addition operations are not completely independent because they share the carry-propagate adder that performs the final rounding (as well as the rounding logic).

Addition on the R3010 uses a mixture of ripple, CLA, and carry-select. A carry-select adder is used in the fashion of Figure J.20 (page J-43). Within each half, carries are propagated using a hybrid ripple-CLA scheme of the type indicated in Figure J.19 (page J-42). However, this is further tuned by varying the size of each block, rather than having each fixed at 4 bits (as they are in Figure J.19). The multiplier is midway between the designs of Figures J.2 (page J-4) and J.27 (page J-50). It has an array just large enough so that output can be fed back into the input without having to be clocked. Also, it uses radix-4 Booth recoding and the even/odd technique of Figure J.29 (page J-52). The R3010 can do a divide and multiply in parallel (like the Weitek chip but unlike the TI chip). The divider is a radix-4 SRT method with quotient digits -2, -1, 0, 1, and 2, and is similar to that described in Taylor [1985]. Double-precision division is about four times slower than multiplication. The R3010 shows that for chips using an $0(n)$ multiplier, an SRT divider can operate fast enough to keep a reasonable ratio between multiply and divide.

The Weitek 3364 has independent add, multiply, and divide units. It also uses radix-4 SRT division. However, the add and multiply operations on the Weitek chip are pipelined. The three addition stages are (1) exponent compare, (2) add followed by shift (or *vice versa*), and (3) final rounding. Stages (1) and (3) take only a half-cycle, allowing the whole operation to be done in two cycles, even though there are three pipeline stages. The multiplier uses an array of the style of Figure J.28 but uses radix-8 Booth recoding, which means it must compute 3 times the multiplier. The three multiplier pipeline stages are (1) compute $3b$, (2) pass through array, and (3) final carry-propagation add and round. Single precision passes through the array once, double precision twice. Like addition, the latency is two cycles.

The Weitek chip uses an interesting addition algorithm. It is a variant on the carry-skip adder pictured in Figure J.18 (page J-42). However, P_{ij}, which is the logical AND of many terms, is computed by rippling, performing one AND per ripple. Thus, while the carries propagate left within a block, the value of P_{ij} is propagating right within the next block, and the block sizes are chosen so that both waves complete at the same time. Unlike the MIPS chip, the 3364 has hardware square root, which shares the divide hardware. The ratio of double-precision multiply to divide is 2:17. The large disparity between multiply and divide is due to the fact that multiplication uses radix-8 Booth recoding, while division uses a radix-4 method. In the MIPS R3010, multiplication and division use the same radix.

The notable feature of the TI 8847 is that it does division by iteration (using the Goldschmidt algorithm discussed in Section J.6). This improves the speed of division (the ratio of multiply to divide is 3:11), but means that multiplication and division cannot be done in parallel as on the other two chips. Addition has a two-stage pipeline. Exponent compare, fraction shift, and fraction addition are done in the first stage, normalization and rounding in the second stage. Multiplication uses a binary tree of signed-digit adders and has a three-stage pipeline. The first stage passes through the array, retiring half the bits; the second stage passes through the array a second time; and the third stage converts from signed-digit form to two's complement. Since there is only one array, a new multiply operation can only be initiated in every other cycle. However, by slowing down the clock, two passes through the array can be made in a single cycle. In this case, a new multiplication can be initiated in each cycle. The 8847 adder uses a carry-select algorithm rather than carry-lookahead. As mentioned in Section J.6, the TI carries 60 bits of precision in order to do correctly rounded division.

These three chips illustrate the different trade-offs made by designers with similar constraints. One of the most interesting things about these chips is the diversity of their algorithms. Each uses a different add algorithm, as well as a different multiply algorithm. In fact, Booth recoding is the only technique that is universally used by all the chips.

J.11 Fallacies and Pitfalls

Fallacy *Underflows rarely occur in actual floating-point application code*

Although most codes rarely underflow, there are actual codes that underflow frequently. SDRWAVE [Kahaner 1988], which solves a one-dimensional wave equation, is one such example. This program underflows quite frequently, even when functioning properly. Measurements on one machine show that adding hardware support for gradual underflow would cause SDRWAVE to run about 50% faster.

Fallacy *Conversions between integer and floating point are rare*

In fact, in spice they are as frequent as divides. The assumption that conversions are rare leads to a mistake in the SPARC version 8 instruction set, which does not provide an instruction to move from integer registers to floating-point registers.

Pitfall *Don't increase the speed of a floating-point unit without increasing its memory bandwidth*

A typical use of a floating-point unit is to add two vectors to produce a third vector. If these vectors consist of double-precision numbers, then each floating-point add will use three operands of 64 bits each, or 24 bytes of memory. The memory bandwidth requirements are even greater if the floating-point unit can perform addition and multiplication in parallel (as most do).

Pitfall *$-x$ is not the same as $0-x$*

This is a fine point in the IEEE standard that has tripped up some designers. Because floating-point numbers use the sign magnitude system, there are two zeros, +0 and −0. The standard says that $0-0=+0$, whereas $-(0)=-0$. Thus, $-x$ is not the same as $0-x$ when $x=0$.

J.12 Historical Perspective and References

The earliest computers used fixed point rather than floating point. In “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” Burks, Goldstine, and von Neumann [1946] put it like this:

> *There appear to be two major purposes in a “floating” decimal point system both of which arise from the fact that the number of digits in a word is a constant fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem “scale factors”—multiplicative constants which serve to keep numbers within the limits of the machine.*
>
> *There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity that could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

This enables us to see things from the perspective of early computer designers, who believed that saving computer time and memory were more important than saving programmer time.

The original papers introducing the Wallace tree, Booth recoding, SRT division, overlapped triplets, and so on are reprinted in Swartzlander [1990]. A good explanation of an early machine (the IBM 360/91) that used a pipelined Wallace tree, Booth recoding, and iterative division is in Anderson et al. [1967]. A discussion of the average time for single-bit SRT division is in Freiman [1961]; this is one of the few interesting historical papers that does not appear in Swartzlander.

The standard book of Mead and Conway [1980] discouraged the use of CLAs as not being cost effective in VLSI. The important paper by Brent and Kung [1982] helped combat that view. An example of a detailed layout for CLAs can be found in Ngai and Irwin [1985] or in Weste and Eshraghian [1993], and a more theoretical treatment is given by Leighton [1992]. Takagi, Yasuura, and Yajima [1985] provide a detailed description of a signed-digit tree multiplier.

Before the ascendancy of IEEE arithmetic, many different floating-point formats were in use. Three important ones were used by the IBM 370, the DEC VAX, and the Cray. Here is a brief summary of these older formats. The VAX format is closest to the IEEE standard. Its single-precision format (F format) is like IEEE single precision in that it has a hidden bit, 8 bits of exponent, and 23 bits of fraction. However, it does not have a sticky bit, which causes it to round halfway cases up instead of to even. The VAX has a slightly different exponent range from IEEE single: $E_{\min}$ is -128 rather than -126 as in IEEE, and $E_{\max}$ is 126 instead of 127. The main differences between VAX and IEEE are the lack of special values and gradual underflow. The VAX has a reserved operand, but it works like a signaling NaN: It traps whenever it is referenced. Originally, the VAX's double precision (D format) also had 8 bits of exponent. However, as this is too small for many applications, a G format was added; like the IEEE standard, this format has 11 bits of exponent. The VAX also has an H format, which is 128 bits long.

The IBM 370 floating-point format uses base 16 rather than base 2. This means it cannot use a hidden bit. In single precision, it has 7 bits of exponent and 24 bits (6 hex digits) of fraction. Thus, the largest representable number is $16^{27} = 2^4 \times 2^7 = 2^{29}$, compared with 2^{28} for IEEE. However, a number that is normalized in the hexadecimal sense only needs to have a nonzero leading digit. When interpreted in binary, the three most-significant bits could be zero. Thus, there are potentially fewer than 24 bits of significance. The reason for using the higher base was to minimize the amount of shifting required when adding floating-point numbers. However, this is less significant in current machines, where the floating-point add time is usually fixed independently of the operands. Another difference between 370 arithmetic and IEEE arithmetic is that the 370 has neither a round digit nor a sticky digit, which effectively means that it truncates rather than rounds. Thus, in many computations, the result will systematically be too small. Unlike the VAX and IEEE arithmetic, every bit pattern is a valid number. Thus, library routines must establish conventions for what to return in case of errors. In the IBM FORTRAN library, for example, $\sqrt{-4}$ returns 2!

Arithmetic on Cray computers is interesting because it is driven by a motivation for the highest possible floating-point performance. It has a 15-bit exponent field and a 48-bit fraction field. Addition on Cray computers does not have a guard digit, and multiplication is even less accurate than addition. Thinking of multiplication as a sum of p numbers, each $2p$ bits long, Cray computers drop the low-order bits of each summand. Thus, analyzing the exact error characteristics of the multiply operation is not easy. Reciprocals are computed using iteration, and division of a by b is done by multiplying a times $1/b$. The errors in multiplication and reciprocation combine to make the last three bits of a divide operation unreliable. At least Cray computers serve to keep numerical analysts on their toes!

The IEEE standardization process began in 1977, inspired mainly by W. Kahan and based partly on Kahan's work with the IBM 7094 at the University of Toronto [Kahan 1968]. The standardization process was a lengthy affair, with gradual underflow causing the most controversy. (According to Cleve Moler, visitors to the United States were advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.) The standard was finally approved in 1985. The Intel 8087 was the first major commercial IEEE implementation and appeared in 1981, before the standard was finalized. It contains features that were eliminated in the final standard, such as projective bits. According to Kahan, the length of double-extended precision was based on what could be implemented in the 8087. Although the IEEE standard was not based on any existing floating-point system, most of its features were present in some other system. For example, the CDC 6600 reserved special bit patterns for INDEFINITE and INFINITY, while the idea of denormal numbers appears in Goldberg [1967] as well as in Kahan [1968]. Kahan was awarded the 1989 Turing prize in recognition of his work on floating point.

Although floating point rarely attracts the interest of the general press, newspapers were filled with stories about floating-point division in November 1994. A bug in the division algorithm used on all of Intel's Pentium chips had just come to light. It was discovered by Thomas Nicely, a math professor at Lynchburg College in Virginia. Nicely found the bug when doing calculations involving reciprocals of prime numbers. News of Nicely's discovery first appeared in the press on the front page of the November 7 issue of *Electronic Engineering Times.* Intel's immediate response was to stonewall, asserting that the bug would only affect theoretical mathematicians. Intel told the press, "This doesn't even qualify as an errata … even if you're an engineer, you're not going to see this."

Under more pressure, Intel issued a white paper, dated November 30, explaining why they didn't think the bug was significant. One of their arguments was based on the fact that if you pick two floating-point numbers at random and divide one into the other, the chance that the resulting quotient will be in error is about 1 in 9 billion. However, Intel neglected to explain why they thought that the typical customer accessed floating-point numbers randomly.

Pressure continued to mount on Intel. One sore point was that Intel had known about the bug before Nicely discovered it, but had decided not to make it public. Finally, on December 20, Intel announced that they would unconditionally replace any Pentium chip that used the faulty algorithm and that they would take an unspecified charge against earnings, which turned out to be $300 million.

The Pentium uses a simple version of SRT division as discussed in Section J.9. The bug was introduced when they converted the quotient lookup table to a PLA. Evidently there were a few elements of the table containing the quotient digit 2 that Intel thought would never be accessed, and they optimized the PLA design using this assumption. The resulting PLA returned 0 rather than 2 in these situations. However, those entries were really accessed, and this caused the division bug. Even though the effect of the faulty PLA was to cause 5 out of 2048 table entries to be wrong, the Pentium only computes an incorrect quotient 1 out of 9 billion times on random inputs. This is explored in Exercise J.34.

References

Anderson, S.F., Earle, J.G., Goldschmidt, R.E., Powers, D.M., 1967. The IBM System/360 Model 91: Floating-point execution unit. IBM J. Research and Development 11, 34–53. Reprinted in Swartzlander [1990]. *Good description of an early high-performance floating-point unit that used a pipelined Wallace tree multiplier and iterative division.*

Bell, C.G., Newell, A., 1971. Computer Structures: Readings and Examples. McGraw-Hill, New York.

Birman, M., Samuels, A., Chu, G., Chuk, T., Hu, L., McLeod, J., Barnes, J., 1990. Developing the WRL3170/3171 SPARC floating-point coprocessors. IEEE Micro 10 (1), 55–64. *These chips have the same floating-point core as the Weitek 3364, and this paper has a fairly detailed description of that floating-point design.*

Brent, R.P., Kung, H.T., 1982. A regular layout for parallel adders. IEEE Trans. on Computers C-31, 260–264. *This is the paper that popularized CLAs in VLSI.*

Burgess, N., Williams, T., 1995. Choices of operand truncation in the SRT division algorithm. IEEE Trans. on Computers 44, 7. *Analyzes how many bits of divisor and remainder need to be examined in SRT division.*

Burks, A.W., Goldstine, H.H., von Neumann, J., 1946. Preliminary discussion of the logical design of an electronic computing instrument. In: Aspray, W., Burks, A. (Eds.), Papers of John von Neumann. Report to the U.S. Army Ordnance Department, p. 1; also appears. MIT Press, Cambridge, Mass, pp. 97–146. Tomash Publishers, Los Angeles, 1987.

Cody, W.J., Coonen, J.T., Gay, D.M., Hanson, K., Hough, D., Kahan, W., Karpinski, R., Palmer, J., Ris, F.N., Stevenson, D., 1984. A proposed radix- and word-length-independent standard for floating-point arithmetic. IEEE Micro 4 (4), 86–100. *Contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754. However, be aware that there are some differences between this draft and the final standard.*

Coonen, J., 1984. Contributions to a proposed standard for binary floating point arithmetic. Ph.D. thesis. University of California–Berkeley. *The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversion.*

Darley, H.M., et al., 1989. Floating point/integer processor with divide and square root functions. U.S. Patent. 4 (878,190) October 31, 1989. *Pretty readable as patents go. Gives a high-level view of the TI 8847 chip, but doesn't have all the details of the division algorithm.*

Demmel, J.W., Li, X., 1994. Faster numerical algorithms via exception handling. IEEE Trans. on Computers 43 (8), 983–992. *A good discussion of how the features unique to IEEE floating point can improve the performance of an important software library.*

Freiman, C.V., 1961. Statistical analysis of certain binary division algorithms. Proc. IRE 49 (1), 91–103. *Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.*

Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. Computing Surveys 23 (1), 5–48. *Contains an in-depth tutorial on the IEEE standard from the software point of view.*

Goldberg, I.B., 1967. 27 bits are not enough for 8-digit accuracy. Comm. ACM 10 (2), 105–106. *This paper proposes using hidden bits and gradual underflow.*

Gosling, J.B., 1980. Design of Arithmetic Units for Digital Computers. Springer-Verlag, New York. *A concise, well-written book, although it focuses on MSI designs.*

Hamacher, V.C., Vranesic, Z.G., Zaky, S.G., 1984. Computer Organization, 2nd ed. McGraw-Hill, New York. *Introductory computer architecture book with a good chapter on computer arithmetic.*

Hwang, K., 1979. Computer Arithmetic: Principles, Architecture, and Design. Wiley, New York. *This book contains the widest range of topics of the computer arithmetic books.*

IEEE, 1985. IEEE standard for binary floating-point arithmetic. SIGPLAN Notices 22 (2), 9–25. *IEEE 754 is reprinted here.*

Kahan, W., 1968. 7094-II system support for numerical analysis. SHARE Secretarial Distribution. SSD-159. *This system had many features that were incorporated into the IEEE floating-point standard.*

Kahaner, D.K., 1988. Benchmarks for ‘real’ programs. SIAM News.(November).. *The benchmark presented in this article turns out to cause many underflows.*

Knuth, D., 1981. 2nd ed. The Art of Computer Programming.Vol. II. Addison-Wesley, Reading, Mass. *Has a section on the distribution of floating-point numbers.*

Kogge, P., 1981. The Architecture of Pipelined Computers. McGraw-Hill, New York. *Has a brief discussion of pipelined multipliers.*

Kohn, L., Fu, S.-W., 1989. A 1,000,000 transistor microprocessor. In: IEEE Int'l. Solid-State Circuits Conf. Digest of Technical Papers, pp. 54–55. *There are several articles about the i860, but this one contains the most details about its floating-point algorithms.*

Koren, I., 1989. Computer Arithmetic Algorithms. Prentice Hall, Englewood Cliffs, N.J..

Leighton, F.T., 1992. Introduction to Parallel Algorithms and Architectures: Arrays. Trees, Hypercubes, Morgan Kaufmann, San Francisco. *This is an excellent book, with emphasis on the complexity analysis of algorithms. Section 1.2.1 has a nice discussion of carry-lookahead addition on a tree.*

Magenheimer, D.J., Peters, L., Pettis, K.W., Zuras, D., 1988. Integer multiplication and division on the HP Precision architecture. IEEE Trans. on Computers 37 (8), 980–990. *Gives rationale for the integer- and divide-step instructions in the Precision architecture.*

Markstein, P.W., 1990. Computation of elementary functions on the IBM RISC System/6000 processor. IBM J. of Research and Development 34 (1), 111–119. *Explains how to use fused muliply-add to compute correctly rounded division and square root.*

Mead, C., Conway, L., 1980. Introduction to VLSI Systems. Addison-Wesley, Reading, Mass.

Montoye, R.K., Hokenek, E., Runyon, S.L., 1990. Design of the IBM RISC System/6000 floating-point execution. IBM J. of Research and Development 34 (1), 59–70. *Describes one implementation of fused multiply-add.*

Ngai, T.-F., Irwin, M.J., 1985. Regular, area-time efficient carry-lookahead adders. In: Proc. Seventh IEEE Symposium on Computer Arithmetic, pp. 9–15. *Describes a CLA like that of Figure J.17, where the bits flow up and then come back down.*

Patterson, D.A., Hennessy, J.L., 2009. Computer Organization and Design: The Hardware/Software Interface, 4th Edition Morgan Kaufmann, San Francisco. *Chapter 3 is a gentler introduction to the first third of this appendix.*

Peng, V., Samudrala, S., Gavrielov, M., 1987. On the implementation of shifters, multipliers, and dividers in VLSI floating point units. In: Proc. Eighth IEEE Symposium on Computer Arithmetic, pp. 95–102. *Highly recommended survey of different techniques actually used in VLSI designs.*

Rowen, C., Johnson, M., Ries, P., 1988. The MIPS R3010 floating-point coprocessor. IEEE Micro 53–62 (June).

Santoro, M.R., Bewick, G., Horowitz, M.A., 1989. Rounding algorithms for IEEE multipliers. In: Proc. Ninth IEEE Symposium on Computer Arithmetic, pp. 176–183. *A very readable discussion of how to efficiently implement rounding for floating-point multiplication.*

Scott, N.R., 1985. Computer Number Systems and Arithmetic. Prentice Hall, Englewood Cliffs, N.J.

Swartzlander, E. (Ed.), 1990. Computer Arithmetic. IEEE Computer Society Press, Los Alamitos, Calif. *A collection of historical papers in two volumes.*

Takagi, N., Yasuura, H., Yajima, S., 1985. High-speed VLSI multiplication algorithm with a redundant binary addition tree. IEEE Trans. on Computers C-34 (9), 789–796. *A discussion of the binary tree signed multiplier that was the basis for the design used in the TI 8847.*

Taylor, G.S., 1981. Compatible hardware for division and square root. In: Proc. Fifth IEEE Symposium on Computer Arithmetic, May 18–19, 1981. Ann Arbor, Mich, pp. 127–134. *Good discussion of a radix-4 SRT division algorithm.*

Taylor, G.S., 1985. Radix 16 SRT dividers with overlapped quotient selection stages. In: Proc. Seventh IEEE Symposium on Computer Arithmetic, June 4–6, 1985, pp. 64–71 Urbana, Ill.. *Describes a very sophisticated high-radix division algorithm.*

Weste, N., Eshraghian, K., 1993. Principles of CMOS VLSI Design: A Systems Perspective, 2nd ed. Addison-Wesley, Reading, Mass. *This textbook has a section on the layouts of various kinds of adders.*

Williams, T.E., Horowitz, M., Alverson, R.L., Yang, T.S., 1987. A self-timed chip for division. In: Advanced Research in VLSI, Proc. 1987 Stanford Conf. MIT Press, Cambridge, Mass. *Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.*

Exercises

J.1 [12] < J.2 > Using n bits, what is the largest and smallest integer that can be represented in the two's complement system?

J.2 [20/25] < J.2 > In the subsection "Signed Numbers" (page J-7), it was stated that two's complement overflows when the carry into the high-order bit position is different from the carry-out from that position.

a. [20] <J.2> Give examples of pairs of integers for all four combinations of carry-in and carry-out. Verify the rule stated above.

b. [25] <J.2> Explain why the rule is always true.

J.3 [12] <J.2> Using 4-bit binary numbers, multiply -8×-8 using Booth recoding.

J.4 [15] <J.2> Equations J.2.1 and J.2.2 are for adding two n-bit numbers. Derive similar equations for subtraction, where there will be a borrow instead of a carry.

J.5 [25] <J.2> On a machine that doesn't detect integer overflow in hardware, show how you would detect overflow on a signed addition operation in software.

J.6 [15/15/20] <J.3> Represent the following numbers as single-precision and double-precision IEEE floating-point numbers:

a. [15] <J.3> 10.

b. [15] <J.3> 10.5.

c. [20] <J.3> 0.1.

J.7 [12/12/12/12/12] <J.3> Below is a list of floating-point numbers. In single precision, write down each number in binary, in decimal, and give its representation in IEEE arithmetic.

a. [12] <J.3> The largest number less than 1.

b. [12] <J.3> The largest number.

c. [12] <J.3> The smallest positive normalized number.

d. [12] <J.3> The largest denormal number.

e. [12] <J.3> The smallest positive number.

J.8 [15] <J.3> Is the ordering of nonnegative floating-point numbers the same as integers when denormalized numbers are also considered?

J.9 [20] <J.3> Write a program that prints out the bit patterns used to represent floating-point numbers on your favorite computer. What bit pattern is used for NaN?

J.10 [15] <J.4> Using $p = 4$, show how the binary floating-point multiply algorithm computes the product of 1.875×1.875.

J.11 [12/10] <J.4> Concerning the addition of exponents in floating-point multiply:

a. [12] <J.4> What would the hardware that implements the addition of exponents look like?

b. [10] <J.4> If the bias in single precision were 129 instead of 127, would addition be harder or easier to implement?

J.12 [15/12] <J.4> In the discussion of overflow detection for floating-point multiplication, it was stated that (for single precision) you can detect an overflowed exponent by performing exponent addition in a 9-bit adder.

a. [15] < J.4 > Give the exact rule for detecting overflow.

b. [12] < J.4 > Would overflow detection be any easier if you used a 10-bit adder instead?

J.13 [15/10] < J.4 > Floating-point multiplication:

a. [15] < J.4 > Construct two single-precision floating-point numbers whose product doesn't overflow until the final rounding step.

b. [10] < J.4 > Is there any rounding mode where this phenomenon cannot occur?

J.14 [15] < J.4 > Give an example of a product with a denormal operand but a normalized output. How large was the final shifting step? What is the maximum possible shift that can occur when the inputs are double-precision numbers?

J.15 [15] < J.5 > Use the floating-point addition algorithm on page J-23 to compute $1.010_2 - .1001_2$ (in 4-bit precision).

J.16 [10/15/20/20/20] < J.5 > In certain situations, you can be sure that $a+b$ is exactly representable as a floating-point number, that is, no rounding is necessary.

a. [10] < J.5 > If a, b have the same exponent and different signs, explain why $a+b$ is exact. This was used in the subsection "Speeding Up Addition" on page J-25.

b. [15] < J.5 > Give an example where the exponents differ by 1, a and b have different signs, and $a+b$ is not exact.

c. [20] < J.5 > If $a \geq b \geq 0$, and the top two bits of a cancel when computing $a-b$, explain why the result is exact (this fact is mentioned on page J-22).

d. [20] < J.5 > If $a \geq b \geq 0$, and the exponents differ by 1, show that $a-b$ is exact unless the high order bit of $a-b$ is in the same position as that of a (mentioned in "Speeding Up Addition," page J-25).

e. [20] < J.5 > If the result of $a-b$ or $a+b$ is denormal, show that the result is exact (mentioned in the subsection "Underflow," on page J-36).

J.17 [15/20] < J.5 > Fast floating-point addition (using parallel adders) for $p=5$.

a. [15] < J.5 > Step through the fast addition algorithm for $a+b$, where $a=1.0111_2$ and $b=.11011_2$.

b. [20] < J.5 > Suppose the rounding mode is toward $+\infty$. What complication arises in the above example for the adder that assumes a carry-out? Suggest a solution.

J.18 [12] < J.4, J.5 > How would you use two parallel adders to avoid the final round-up addition in floating-point multiplication?

J.19 [30/10] < J.5 > This problem presents a way to reduce the number of addition steps in floating-point addition from three to two using only a single adder.

a. [30] < J.5 > Let A and B be integers of opposite signs, with a and b their magnitudes. Show that the following rules for manipulating the unsigned numbers a and b gives $A+B$.

1. Complement one of the operands.
2. Use end-around carry to add the complemented operand and the other (uncomplemented) one.
3. If there was a carry-out, the sign of the result is the sign associated with the uncomplemented operand.
4. Otherwise, if there was no carry-out, complement the result, and give it the sign of the complemented operand.

b. [10] <J.5> Use the above to show how steps 2 and 4 in the floating-point addition algorithm on page J-23 can be performed using only a single addition.

J.20 [20/15/20/15/20/15] <J.6> Iterative square root.

a. [20] <J.6> Use Newton's method to derive an iterative algorithm for square root. The formula will involve a division.

b. [15] <J.6> What is the fastest way you can think of to divide a floating-point number by 2?

c. [20] <J.6> If division is slow, then the iterative square root routine will also be slow. Use Newton's method on $f(x) = 1/x^2 - a$ to derive a method that doesn't use any divisions.

d. [15] <J.6> Assume that the ratio division by 2 : floating-point add : floating-point multiply is 1:2:4. What ratios of multiplication time to divide time makes each iteration step in the method of part (c) faster than each iteration in the method of part (a)?

e. [20] <J.6> When using the method of part (a), how many bits need to be in the initial guess in order to get double-precision accuracy after three iterations? (You may ignore rounding error.)

f. [15] <J.6> Suppose that when spice runs on the TI 8847, it spends 16.7% of its time in the square root routine (this percentage has been measured on other machines). Using the values in Figure J.36 and assuming three iterations, how much slower would spice run if square root were implemented in software using the method of part(a)?

J.21 [10/20/15/15/15] <J.6> Correctly rounded iterative division. Let a and b be floating-point numbers with p-bit significands ($p = 53$ in double precision). Let q be the exact quotient $q = a/b$, $1 \leq q < 2$. Suppose that $\overline{q}$ is the result of an iteration process, that $\overline{q}$ has a few extra bits of precision, and that $0 < q - \overline{q} < 2^{-p}$. For the following, it is important that $\overline{q} < q$, even when q can be exactly represented as a floating-point number.

a. [10] <J.6> If x is a floating-point number, and $1 \leq x < 2$, what is the next representable number after x?

b. [20] <J.6> Show how to compute q' from $\overline{q}$, where q' has $p + 1$ bits of precision and $|q - q'| < 2^{-p}$.

c. [15] <J.6> Assuming round to nearest, show that the correctly rounded quotient is either q', $q' - 2^{-p}$, or $q' + 2^{-p}$.

d. [15] < J.6 > Give rules for computing the correctly rounded quotient from q' based on the low-order bit of q' and the sign of $a - bq'$.

e. [15] < J.6 > Solve part (c) for the other three rounding modes.

J.22 [15] < J.6 > Verify the formula on page J-30. (*Hint*: If $x_n = x_0(2 - x_0 b) \times \Pi_{i=1,n}\left[1 + (1 - x_0 b)^{2^i}\right]$, then $2 - x_n b = 2 - x_0 b(2 - x_0 b)\Pi\left[1 + (1 - x_0 b)^{2^i}\right] = 2 - \left[1 - (1 - x_0 b)^2\right]\Pi\left[1 + (1 - x_0 b)^{2^i}\right]$.)

J.23 [15] < J.7 > Our example that showed that double rounding can give a different answer from rounding once used the round-to-even rule. If halfway cases are always rounded up, is double rounding still dangerous?

J.24 [10/10/20/20] < J.7 > Some of the cases of the italicized statement in the "Precisions" subsection (page J-33) aren't hard to demonstrate.

a. [10] < J.7 > What form must a binary number have if rounding to q bits followed by rounding to p bits gives a different answer than rounding directly to p bits?

b. [10] < J.7 > Show that for multiplication of p-bit numbers, rounding to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \geq 2p$.

c. [20] < J.7 > If a and b are p-bit numbers with the same sign, show that rounding $a + b$ to q bits followed by rounding to p bits is the same as rounding immediately to p bits if $q \geq 2p + 1$.

d. [20] < J.7 > Do part (c) when a and b have opposite signs.

J.25 [Discussion] < J.7 > In the MIPS approach to exception handling, you need a test for determining whether two floating-point operands could cause an exception. This should be fast and also not have too many false positives. Can you come up with a practical test? The performance cost of your design will depend on the distribution of floating-point numbers. This is discussed in Knuth [1981] and the Hamming paper in Swartzlander [1990].

J.26 [12/12/10] < J.8 > Carry-skip adders.

a. [12] < J.8 > Assuming that time is proportional to logic levels, how long does it take an n-bit adder divided into (fixed) blocks of length k bits to perform an addition?

b. [12] < J.8 > What value of k gives the fastest adder?

c. [10] < J.8 > Explain why the carry-skip adder takes time $0(\sqrt{n})$.

J.27 [10/15/20] < J.8 > Complete the details of the block diagrams for the following adders.

a. [10] < J.8 > In Figure J.15, show how to implement the "1" and "2" boxes in terms of AND and OR gates.

b. [15] < J.8 > In Figure J.19, what signals need to flow from the adder cells in the top row into the "C" cells? Write the logic equations for the "C" box.

c. [20] < J.8 > Show how to extend the block diagram in J.17 so it will produce the carry-out bit c_8.

J.28 [15] <J.9> For ordinary Booth recoding, the multiple of b used in the ith step is simply $a_{i-1} - a_i$. Can you find a similar formula for radix-4 Booth recoding (overlapped triplets)?

J.29 [20] <J.9> Expand Figure J.29 in the fashion of J.27, showing the individual adders.

J.30 [25] <J.9> Write out the analog of Figure J.25 for radix-8 Booth recoding.

J.31 [18] <J.9> Suppose that $a_{n-1} \ldots a_1 a_0$ and $b_{n-1} \ldots b_1 b_0$ are being added in a signed-digit adder as illustrated in the example on page J-53. Write a formula for the ith bit of the sum, s_i, in terms of a_i, a_{i-1}, a_{i-2}, b_i, b_{i-1}, and b_{i-2}.

J.32 [15] <J.9> The text discussed radix-4 SRT division with quotient digits of -2, $-1, 0, 1, 2$. Suppose that 3 and -3 are also allowed as quotient digits. What relation replaces $|r_i| \leq 2b/3$?

J.33 [25/20/30] <J.9> Concerning the SRT division table, Figure J.34:

a. [25] <J.9> Write a program to generate the results of Figure J.34.

b. [20] <J.9> Note that Figure J.34 has a certain symmetry with respect to positive and negative values of P. Can you find a way to exploit the symmetry and only store the values for positive P?

c. [30] <J.9> Suppose a carry-save adder is used instead of a propagate adder. The input to the quotient lookup table will be k bits of divisor and l bits of remainder, where the remainder bits are computed by summing the top l bits of the sum and carry registers. What are k and l? Write a program to generate the analog of Figure J.34.

J.34 [12/12/12] <J.9, J.12> The first several million Pentium chips produced had a flaw that caused division to sometimes return the wrong result. The Pentium uses a radix-4 SRT algorithm similar to the one illustrated in the example on page J-56 (but with the remainder stored in carry-save format; see Exercise J.33(c)). According to Intel, the bug was due to five incorrect entries in the quotient lookup table.

a. [12] <J.9, J.12> The bad entries should have had a quotient of plus or minus 2, but instead had a quotient of 0. Because of redundancy, it's conceivable that the algorithm could "recover" from a bad quotient digit on later iterations. Show that this is not possible for the Pentium flaw.

b. [12] <J.9, J.12> Since the operation is a floating-point divide rather than an integer divide, the SRT division algorithm on page J-45 must be modified in two ways. First, step 1 is no longer needed, since the divisor is already normalized. Second, the very first remainder may not satisfy the proper bound ($|r| \leq 2b/3$ for Pentium; see page J-55). Show that skipping the very first left shift in step 2(a) of the SRT algorithm will solve this problem.

c. [12] <J.9, J.12> If the faulty table entries were indexed by a remainder that could occur at the very first divide step (when the remainder is the divisor), random testing would quickly reveal the bug. This didn't happen. What does that tell you about the remainder values that index the faulty entries?

J.35 [12] < J.6, J.9 > The discussion of the remainder-step instruction assumed that division was done using a bit-at-a-time algorithm. What would have to change if division were implemented using a higher-radix method?

J.36 [25] < J.9 > In the array of Figure J.28, the fact that an array can be pipelined is not exploited. Can you come up with a design that feeds the output of the bottom CSA into the bottom CSAs instead of the top one, and that will run faster than the arrangement of Figure J.28?

K.1 Introduction K-2
K.2 A Survey of RISC Architectures for Desktop, Server, and Embedded Computers K-3
K.3 The Intel 80x86 K-30
K.4 The VAX Architecture K-50
K.5 The IBM 360/370 Architecture for Mainframe Computers K-69
K.6 Historical Perspective and References K-75

K

Survey of Instruction Set Architectures

RISC: any computer announced after 1985.

Steven Przybylski
A Designer of the Stanford MIPS

K.1 Introduction

This appendix covers 10 instruction set architectures, some of which remain a vital part of the IT industry and some of which have retired to greener pastures. We keep them all in part to show the changes in fashion of instruction set architecture over time.

We start with eight RISC architectures, using RISC V as our basis for comparison. There are billions of dollars of computers shipped each year for ARM (including Thumb-2), MIPS (including microMIPS), Power, and SPARC. ARM dominates in both the PMD (including both smart phones and tablets) and the embedded markets.

The 80x86 remains the highest dollar-volume ISA, dominating the desktop and the much of the server market. The 80x86 did not get traction in either the embedded or PMD markets, and has started to lose ground in the server market. It has been extended more than any other ISA in this book, and there are no plans to stop it soon. Now that it has made the transition to 64-bit addressing, we expect this architecture to be around, although it may play a smaller role in the future then it did in the past 30 years.

The VAX typifies an ISA where the emphasis was on code size and offering a higher level machine language in the hopes of being a better match to programming languages. The architects clearly expected it to be implemented with large amounts of microcode, which made single chip and pipelined implementations more challenging. Its successor was the Alpha, a RISC architecture similar to MIPS and RISC V, but which had a short life.

The vulnerable IBM 360/370 remains a classic that set the standard for many instruction sets to follow. Among the decisions the architects made in the early 1960s were:

- 8-bit byte
- Byte addressing
- 32-bit words
- 32-bit single precision floating-point format + 64-bit double precision floating-point format
- 32-bit general-purpose registers, separate 64-bit floating-point registers
- Binary compatibility across a family of computers with different cost-performance
- Separation of architecture from implementation

As mentioned in Chapter 2, the IBM 370 was extended to be virtualizable, so it had the lowest overhead for a virtual machine of any ISA. The IBM 360/370 remains the foundation of the IBM mainframe business in a version that has extended to 64 bits.

K.2 A Survey of RISC Architectures for Desktop, Server, and Embedded Computers

Introduction

We cover two groups of Reduced Instruction Set Computer (RISC) architectures in this section. The first group is the desktop, server RISCs, and PMD processors:

- Advanced RISC Machines ARMv8, AArch64, the 64-bit ISA,
- MIPS64, version 6, the most recent the 64-bit ISA,
- Power version 3.0, which merges the earlier IBM Power architecture and the PowerPC architecture.
- RISC-V, specifically RV64G, the 64-bit extension of RISC-V.
- SPARCv9, the 64-bit ISA.

As Figure K.1 shows these architectures are remarkably similar.

There are two other important historical RISC processors that are almost identical to those in the list above: the DEC Alpha processor, which was made by Digital Equipment Corporation from 1992 to 2004 and is almost identical to MIPS64. Hewlett-Packard's PA-RISC was produced by HP from about 1986 to 2005, when it was replaced by Itanium. PA-RISC is most closely related to the Power ISA, which emerged from the IBM Power design, itself a descendant of IBM 801.

The second group is the embedded RISCs designed for lower-end applications:

- Advanced RISC Machines, Thumb-2: an 32-bit instruction set with 16-bit and 32-bit instructions. The architecture includes features from both ARMv7 and ARMv8.
- microMIPS64: a version of the MIPS64 instruction set with 16-it instructions, and
- RISC-V Compressed extension (RV64GC), a set of 16-bit instructions added to RV64G

Both RV64GC and microMIPS64 have corresponding 32-bit versions: RV32GC and microMIPS32.

Since the comparison of the base 32-bit or 64-bit desktop and server architecture will examine the differences among those ISAs, our discussion of the embedded architectures focuses on the 16-bit instructions. Figure K.2 shows that these embedded architectures are also similar. In all three, the 16-bit instructions are versions of 32-bit instructions, typically with a restricted set of registers.The idea is to reduce the code size by replacing common 32-bit instructions with 16-bit versions. For RV32GC or Thumb-2, including the 16-bit instructions yields a reduction in code size to about 0.73 of the code size using only the 32-bit ISA (either RV32G or ARMv7).

	ARMv8	**MIPS64 R6**	**Power v3.0**	**RV64G**	**SPARCv9**
Original date (base ISA)	1986	1986	1990	2016	1987
Date of this ISA	2011	2014	2013	2016	2008
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits (flat)	64 bits, flat	64 bits, flat	64 bits, flat	64 bits, flat
Data alignment	Aligned preferred	Aligned preferred	Unaligned	Aligned preferred	Aligned
Data addressing modes	8 (including scaled, pre/post increment)	1 (+1 for FP only)	4	1	2
Integer registers (number, model, size)	31 GPR x 64, plus stack pointer	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits	31 GPR × 64 bits
Separate floating-point registers	32x32 or 32x64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits	32 × 32 or 32 × 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double

Figure K.1 Summary of the most recent version of five architectures for desktop, server, and PMD use (all had earlier versions). Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure K.29. In ARMv8, register 31 is a 0 (like register 0 in the other architectures), but when it is used in a load or store, it is the current stack pointer, a special purpose register. We can either think of SP-based addressing as a different mode (which is how the assembly mnemonics operate) or as simply a register + offset addressing mode (which is how the instruction is encoded).

	microMIPS64	**RV64GC**	**Thumb-2**
Date announced	2009	2016	2003
Instruction size (bits)	16/32	16/32	16/32
Address space (size, model)	32/64 bits, flat	32/64 bits flat	32 bits, flat
Data alignment	Aligned	Aligned, preferred	Aligned
Data addressing modes	2	1	6
Integer registers (number, model, size)	31 GPR × 64 bits	31 GPR × 64 bits	15 GPR × 32 bits
Integer registers accessible by most 16-bit instructions (which use should specifiers)	8 GPR + SP + GP +RA GPRs: 0, 2-7, 17, or 2-7, 16, 17	8 GPRs + SP GPRs: 8-15	8 GPR + SP × 32 bits

Figure K.2 Summary of three recent architectures for embedded applications. All three use 16-bit extensions of a base instruction set. Except for number of data address modes and a number of instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure K.29. An earlier 16-bit version of the MIPS instruction set, called MIPS16, was created in 1995 and was replaced by microMIPS32 and microMIPS64. The first Thumb architecture had only 16-bit instructions and was created in 1996. Thumb-2 is built primarily on ARMv7, the 32-bit ARM instruction set; it offers 16 registers. RISC-V also defines RV32E, which has only 16 registers, includes the 16-bit instructions, and cannot have floating point. It appears that most implementations for embedded applications opt for RV32C or RV64GC.

A key difference among these three architectures is the structure of the base 32-bit ISA. In the case of RV64GC, the 32-bit instructions are exactly those of RV64G. This is possible because RISC V planned for the 16-it option from the beginning, and branch addresses and jump addresses are specified to 16-it boundaries. In the case of microMIPS64, the base ISA is MIPS64, with one change: branch and jump offsets are interpreted as 16-bit rather than 32-bit aligned. (microMIPS also uses the encoding space that was reserved in MIPS64 for user-defined instruction set extensions; such extensions are not part of the base ISA.)

Thumb-2 uses a slightly different approach. The 32-bit instructions in Thumb-2 are mostly a subset of those in ARMv7; certain features that were dropped in ARMv8 are not included (e.g., conditional execution of most instructions and the ability to write the PC as a GPR). Thumb-2 also includes a few dozen instructions introduced in ARMv8, specifically bit field manipulation, additional system instructions, and synchronization support. Thus, the 32-bit instructions in Thumb-2 constitute a unique ISA.

Earlier versions of the 16-bit instruction sets for MIPS (MIPS16) and ARM (Thumb), took the approach of creating a separate mode, invoked by a procedure call, to transfer control to a code segment that employed only 16-bit instructions.

The 16-bit instruction set was not complete and was only intended for user programs that were code-size critical.

One complication of this description is that some of the older RISCs have been extended over the years. We decided to describe the most recent versions of the architectures: ARMv8 (the 64-bit architecture AArch64), MIPS64 R6, Power v3.0, RV64G, and SPARC v9 for the desktop/server/PMD, and the 16-bit subset of the ISAs for microMIPS64, RV64GC, and Thumb-2.

The remaining sections proceed as follows. After discussing the addressing modes and instruction formats of our RISC architectures, we present the survey of the instructions in five steps:

- Instructions found in the RV64G core, described in Appendix A.
- Instructions not found in the RV64G or RV64GC but found in two or more of the other architectures. We describe and organize these by functionality, e.g. instructions that support extended integer arithmetic.
- Instruction groups unique to ARM, MIPS, Power, or SPARC, organized by function.
- Multimedia extensions of the desktop/server/PMD RISCs
- Digital signal-processing extensions of the embedded RISCs

Although the majority of the instructions in these architectures are included, we have not included every single instruction; this is especially true for the Power and ARM ISAs, which have *many* instructions.

Addressing Modes and Instruction Formats

Figure K.3 shows the data addressing modes supported by the desktop/server/PMD architectures. Since all, but ARM, have one register that always has the value 0 when used in address modes, the absolute address mode with limited range can be synthesized using register 0 as the base in displacement addressing. (This register can be changed by arithmetic-logical unit (ALU) operations in PowerPC, but is always zero when it is used in an address calculation.) Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

As Figure K.4 shows, the embedded architectures restrict the registers that can be accessed with the 16-bit instructions, typically to only 8 registers, for most instructions, and a few special instructions that refer to other registers. Figure K.5 shows the data addressing modes supported by the embedded architectures in their 16-bit instruction mode. These versions of load/store instructions restrict the registers that can be used in address calculations, as well as significantly shorten the immediate fields, used for displacements.

References to code are normally PC-relative, although jump register indirect is supported for returning from procedures, for case statements, and for pointer function calls. One variation is that PC-relative branch addresses are often shifted left 2 bits before being added to the PC for the desktop RISCs, thereby increasing the branch distance. This works because the length of all instructions for the desktop

	ARMv8	**MIPS64 R6**	**Power v3.0**	**RV64G**	**SPARCv9**
Register + offset (displacement or based)	B, H, W, D	B, H, W, D	B, H, W, D	B, H, W, D	B, H, W, D
Register + register (indexed)	B, H, W, D		B, H, W, D		B, H, W, D
Register + scaled register (scaled)	B, H, W, D	W,D			
Register + register + offset	B, H, W, D				
Register + offset & update register to effective address (based with update)	B, H, W, D		B, H, W, D		
Register & update register to register + offset (register with update)	B, H, W, D				
Register + Register & update register to effective address (indexed with update)	B, H, W, D		B, H, W, D		
PC-relative (PC + displacement)	W, D	W, D			

Figure K.3 Summary of data addressing modes supported by the desktop architectures, where B, H, W, D indicate what datatypes can use the addressing mode. Note that ARM includes two different types of address modes with updates, one of which is included in Power.

Register specifier	microMIPS64	RV64GC	Thumb-2
3-bit	2-7,16, 17	8-15	0-7
stack pointer register	29	2	0 (when used in load/store)
global pointer register	28		
return address register	31	1	14
Using special register	stack pointer or global pointer; 5-bit offset	stack pointer; 5-bit offset	stack pointer; 8-bit offset

Figure K.4 Register encodings for the 16-bit subsets of microMIPS64, RV64GC, and Thumb-2, including the core general purpose registers, and special-purpose registers accessible by some instructions.

Addressing mode	microMIPS64	RV64GC	Thumb-2
Register + offset (displacement or based)	4-bit offset, one of 8 registers	5-bit offset, one of 8 registers	5-bit offset, one of 8 registers
PC-relative data			word only; 8-bit offset
Using special register	stack pointer or global pointer; 5-bit offset	stack pointer; 5-bit offset	stack pointer; 8-bit offset

Figure K.5 Summary of data addressing modes supported by the embedded architectures. microMIPS64, RV64c, and Thumb-2 show only the modes supported in 16-bit instruction formats. The stack pointer in RV64GC and microMIPS64 is a designed GPR; it is another version of r31 is Thumb-2. In microMIPS64, the global pointer is register 30 and is used by the linkage convention to point to the global variable data pool. Notice that typically only 8 registers are accessible as base registers (and as we will see as ALU sources and destinations).

RISCs is 32 bits and instructions must be aligned on 32-bit words in memory. Embedded architectures and RISC V (when extended) have 16-bit-long instructions and usually shift the PC-relative address by 1 for similar reasons.

Figure K.6 shows the most important instruction formats of the desktop/server/PMD RISC instructions. Each instruction set architecture uses four primary instruction formats, which typically include 90–98% of the instructions. The register-register format is used for register-register ALU instructions, while the ALU immediate format is used for ALU instructions with an immediate operand and also for loads and stores. The branch format is used for conditional branches, and the jump/call format for unconditional branches (jumps) and procedures calls.

There are a number of less frequently used instruction formats that Figure K.6 leaves out. Figure K.7 summarizes these for the desktop/server/PMD architectures.

Unlike, their 32-bit base architectures, the 16-bit extensions (microMIPS64, RV64GC, and Thumb-2) are focused on minimizing code. As a result, there are a larger number of instruction formats, even though there are far fewer instructions.

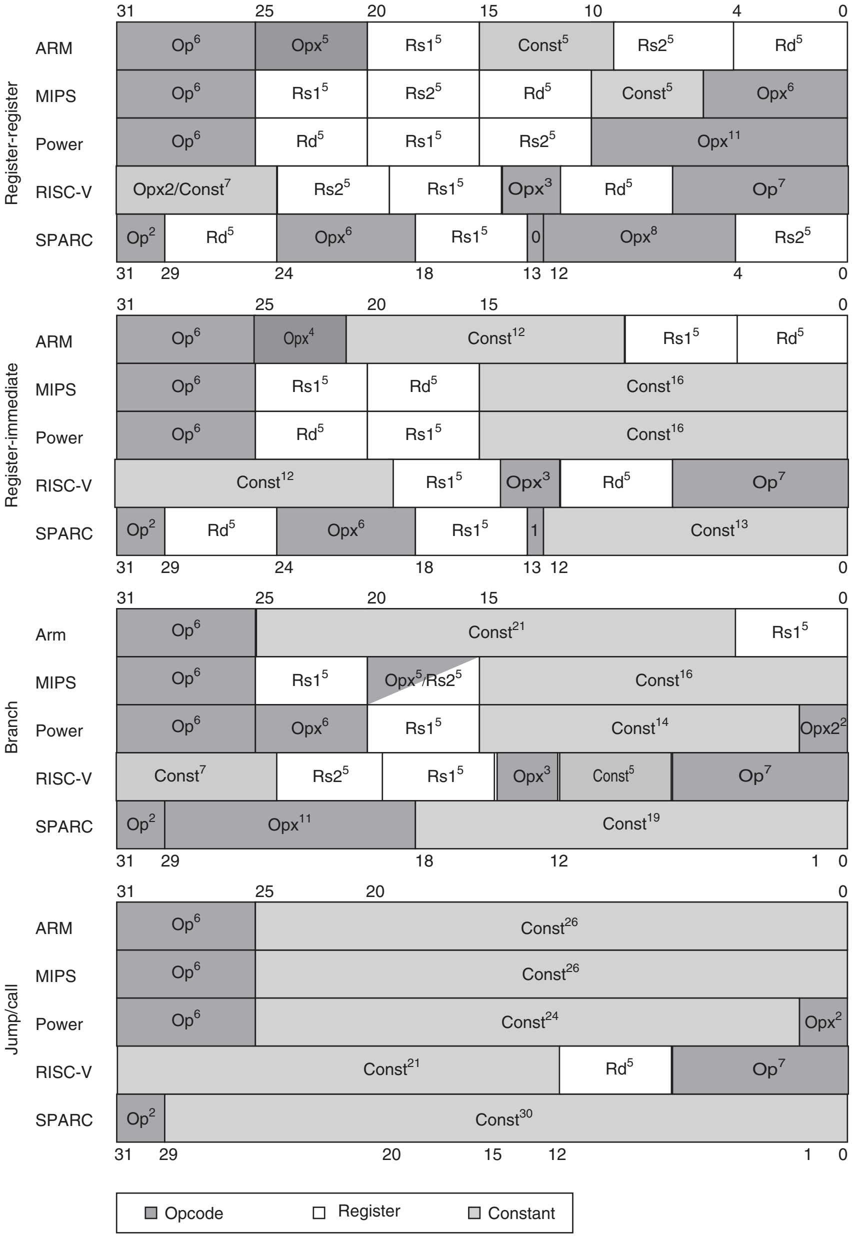

Register-register
31 25 20 15 10 4 0
ARM Op^6 Opx^5 Rs1^5 Const^5 Rs2^5 Rd^5
MIPS Op^6 Rs1^5 Rs2^5 Rd^5 Const^5 Opx^6
Power Op^6 Rd^5 Rs1^5 Rs2^5 Opx^{11}
RISC-V Opx2/Const^7 Rs2^5 Rs1^5 Opx^3 Rd^5 Op^7
SPARC Op^2 Rd^5 Opx^6 Rs1^5 0 Opx^8 Rs2^5
31 29 24 18 13 12 4 0
Register-immediate
31 25 20 15 0
ARM Op^6 Opx^4 Const^{12} Rs1^5 Rd^5
MIPS Op^6 Rs1^5 Rd^5 Const^{16}
Power Op^6 Rd^5 Rs1^5 Const^{16}
RISC-V Const^{12} Rs1^5 Opx^3 Rd^5 Op^7
SPARC Op^2 Rd^5 Opx^6 Rs1^5 1 Const^{13}
31 29 24 18 13 12 0
Branch
31 25 20 15 0
Arm Op^6 Const^{21} Rs1^5
MIPS Op^6 Rs1^5 Opx^5/Rs2^5 Const^{16}
Power Op^6 Opx^6 Rs1^5 Const^{14} Opx2^2
RISC-V Const^7 Rs2^5 Rs1^5 Opx^3 Const^5 Op^7
SPARC Op^2 Opx^{11} Const^{19}
31 29 18 12 1 0
Jump/call
31 25 20 0
ARM Op^6 Const^{26}
MIPS Op^6 Const^{26}
Power Op^6 Const^{24} Opx^2
RISC-V Const^{21} Rd^5 Op^7
SPARC Op^2 Const^{30}
31 29 20 15 12 1 0
▨ Opcode ☐ Register ▨ Constant

Figure K.6 Instruction formats for desktop/server RISC architectures. These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are sometimes scrambled. Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate, address, mask, or sift amount). Although the labels on the instruction formats tell where various instructions are encoded, there are variations. For example, loads and stores, both use the ALU immediate form in MIPS. In RISC-V, loads use the ALU immediate format, while stores use the branch format.

Architecture	Additional instruction formats	Format function and use
ARMv8	At least 10 (many small variations); major forms are shown.	Logical immediates with 13-bit immediate field.
		Shifts with constant amount.(16-bit opcode)
		16-bit immediate form
		Exclusive operations: three register fields
		Branch register: long opcode
		Load/store with address mode bits.
MIPS64	1	A PC-relative set of load/stores using register-immediate format but with 18-bit immediates (since the other source is the PC).
Power	9 (not including a number of small variations or the vector extensions)	DQ-mode: uses the ALU immediate form but takes four bits of the displacement for other functions.
		DS-mode: uses the ALU immediate form but takes two bits of the displacement for other functions.
		DX-form: Like register-immediate, but with a register-source replaced by PC.
		MD, MDS formats: like register-register but used for shifts and rotates.
		X, XS, and several minor variations: used for indexed addressing modes, shifts, and a variety of extended purposes.
		Z22, Z23 formats: used for manipulating floating point numbers
RV64	2	SB format: a variant of the branch format with different immediate treatment
		UJ format: a variant of the jump/call format with different immediate treatment
SPARC	3	Another format for conditional branches containing 3 more bits of displacement (22 total versus 19) but no prediction hints.
		A format with 22-bit immediate used to load the upper half of a register,
		A format for conditional branches based on a register compare with zero.

Figure K.7 Other instruction formats beyond the four major formats of the previous figure. In some cases, there are formats very similar to one of the four core formats, but where a register field is used for other purposes. The Power architecture also includes a number of formats for vector operations.

microMIPs64 and RV64GC have eight and seven major formats, respectively, and Thumb-2 has 15. As Figure K.8 shows, these involve varying number of register operands (0 to 3), different immediate sizes, and even different size register specifiers, with a small number of registers accessible my most instructions, and fewer instructions able to access all 32 registers.

Instructions

The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the RISC-V 64-bit ISA.

Architecture	Opcode main: extended	Register specifiers x length	Immediate field length	Typical instructions
microMIPS64	6	none	10	Jumps
	6	1x5	5	Register-register operation (32 registers) and Load using SP as base register; any destination
	6	1x3	7	Branches equal/not equal zero. Loads using GP. as base.
	6:4	2x3		Register-register operation, rd/rs1, and rs2; 8 registers
	6:1	2x3	3	Register-register immediate, rd/rs1, and rs2; 8 registers
	6	2x3	4	Loads and stores; 8 registers
	6:4	2x3		Register-register operation, rd, and rs1; 8 registers
	6	2x5		Register-register operation; 32 registers.
RV64GC	2:3		11	Jumps
	2:3	1x3	7	Branch
	2:3	1x3	8	Immediate one source register.
	2:3	1x5	6	Store using SP as base.
	2:3	1x5	6	ALU immediate and load using SP as base.
	2:4	2x5		Register-register operation
	2:3	2x3	5	Loads and stores using 8 registers.
Thumb-2	3:2	2x3	5	Shift, move, load/store word/byte
	3:2	1x3	8	immediates: add, subtract, move, and compare
	4:1	1x3	8	Load/store with stack pointer as base, Add to SP or PC, Load/store multiple
	4:3	3x3		Load register indexed
	4:4		8	Conditional branch, system instruction
	4:12			Miscellaneous: 22 different instructions with 12 formats (includes compare and branch on zero, pop/push registers, adjust stack pointer, reverse bytes, IF-THEN instruction).
	5	1x3	8	Load relative to PC
	5		11	Unconditional branch
	6:1	3x3		Add/subtract
	6:3	1x4, 1x3		Special data processing
	6:4	2x3		Logical data processing
	6:6	1x4		Branch and change instruction set (ARM vs. Thumb)

Figure K.8 Instruction formats for the 16-bit instructions of microMIPS64, RV64GC, and Thumb-2. For instructions with a destination and two sources, but only two register fields, the instruction uses one of the registers as both source and destination. Note that the extended opcode field (or function field) and immediate field sometimes overlap or are identical. For RV64GC and microMIPS64, all the formats are shown; for Thumb-2, the Miscellaneous format includes 22 instructions with 12 slightly different formats; we use the extended opcode field, but a few of these instructions have immediate or register fields.

RV64G Core Instructions

Almost every instruction found in the RV64G is found in the other architectures, as Figures K.9 through K.19 show. (For reference, definitions of the RISC-V instructions are found in Section A.9.) Instructions are listed under four categories: data transfer (Figure K.9); arithmetic, logical (Figure K.10); control (Figure K.11 and Figure K.12); and floating point (Figure K.13).

If a RV64G core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figure K.9 through Figure K.13. (To avoid confusion, the destination register will always be the leftmost operand in this appendix, independent of the notation normally used with each architecture.).

Compare and Conditional Branch

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation! Figure K.11 summarizes the control instructions, while Figure K.12 shows details of how conditional branches are handled. SPARC uses the traditional four condition code bits stored in the program status word: *negative, zero, carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using `r0` as the destination. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the EEE 754 conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is =, `not=`, <, <=, >=, `or` <= `0`; three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

Power also uses four condition codes: *less than, greater than, equal*, and *summary overflow*, but it has eight copies of them. This redundancy allows the Power instructions to use different condition codes without conflict, essentially giving Power eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer is followed by a compare to zero that sets the first condition "register." Power also lets the second "register" be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (`CRAND`, `CROR`, `CRXOR`, `CRNAND`, `CRNOR`, `CREQV`), allowing more complex conditions to be tested by a single branch. Finally, Power includes a set of branch count registers, that are automatically decremented when tested, and can be used in a branch condition. There are also special instructions for moving from/to the condition register.

Data transfer (instruction formats)	**R-I**	**R-I**	**R-I, R-R**	**R-I**	**R-I, R-R**
Instruction name	**ARMv8**	**MIPS64**	**Power**	**RV64G**	**SPARC**
Load byte signed/unsigned.	LDR_B	LB_	LBZ; EXTSB	LB_	LD_B
Load halfword signed, unsigned	LDR_H	LH_	LHA/LHZ	LH_	LD_H
Load word	LDRSW/LDR	LW_	LW_	LW_	LD_W
Load double	LDRX	LD	LD	LD	LDD
Load float register SP/DP	LD_	L_C1	LF_	FL_	LD_F
Store byte	STB	SB	STB	SB	STB
Store half word	STW	SH	STH	STH	STH
Store word	STL	SW	STW	SW	ST
Store double word	STX	SD	SD	SD	STD
Store float SP/DP	ST_	S_C1	STF_	FS_	ST_F
Load reserved	LDEXB,LDEXH LDEXW,LDEXD	LL, LLD	lwarx, ldarx,	LR	
Store conditional	STEXB, STEXH, STEXW, STEXD	SC, SCD	stwcx, stdcx	SC	
Read/write spec. register	MF_, MT_	MF, MT_	M_SPR,	csrr_, csrr_i,	RD__,WR__
Move integer to FP register	ITOFS	MFC1/ DMFC1	STW; LDFS	STW; FLDWX	ST; LDF
Move FP to integer register	FTTOIS	MTC1/ DMTC1	STFS; LW	FSTWX; LDW	STF; LD
Synchronize data, instruction stream	DSB ISB	SYNC, SYNCI	SYNC, ISYNC	Fence Fence.i	MEMBAR FLUSH
Atomic operations	LDWAT, LDDAT STWAT, STDAT	LLWP, LLDP, SCWP, SCDP		AMOSWAP.W/D, AMOADD,W.D, AMOAND.W/D, AMOXOR.W/D, AMOOR.W/D, AMOMIN_.W/D, AMOMAX_.W/D	CASA, SWAP, LDSTUB

Figure K.9 Desktop RISC data transfer instructions equivalent to RV64G core. A sequence of instructions to synthesize a RV64G instruction is shown separated by semicolons. The MIPS and Power instructions for atomic operations load and conditionally store a pair of registers and can be used to implement the RV64G atomic operations with at most one intervening ALU instruction. The SPARC instructions: compare-and-swap, swap, LDSTUB provide atomic updates to a memory location and can be used to build the RV64G instructions. The Power3 instructions provide all the functionality, as the RV64G instructions, depending on a function field.

Arithmetic/logical (instruction formats)	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**
Instruction name	**ARM v8**	**MIPS64**	**Power v3**	**RISC-V**	**SPARC v.9**
Add word, immediate	`ADD, ADDI`	`ADDU, ADDUI,`	`ADD, ADDI`	`ADDW, ADDWI`	`ADD`
Add double word	`ADDX`	`DADDU, DADDUI`	`ADD, ADDI`	`ADD, ADDI`	`ADD`
Subtract	`SUB, SUBI`	`SUBU, SUBI`	`SUBF`	`SUBW, SUBWI`	`SUB`
Subtract double word	`SUBX`	`DSUBU, DSUBUI`	`SUBF`	`SUB,SUBI`	`SUB`
Multiply	`MUL, SMUL`	`MUL, MULU, DMUL, DMULU`	`MULLW, MULLI`	`MUL, MULU, MULW, MULWU`	`MULX`
Divide	`MULX, SMULX`	`DIV, DIVU,DDIV, DDIVU`	`DIVW`	`DIV, DIVU, DIVW, DIVWU`	`DIVX`
Remainder		`MOD, MODU, DMOD, DMODU`	`MODSW, MODUW`	`REM, REMU, REMW, REMWU`	
And	`AND, ANDI`	`AND, ANDI`	`AND, ANDI`	`AND, ANDI`	`AND`
Or	`OR, ORI`	`OR, ORI`	`OR, ORI`	`OR, ORI`	`OR`
Xor	`XOR, XORI`	`XOR, XORI`	`XOR, XORI`	`XOR, XORI`	`XOR`
Load bits 31..16	`MOV`	`LUI`	`ADDIS`	`ADDIS`	`SETHI (Bfmt.)`
Load upper bits of PC	`ADR`	`ADDIUPC`	`ADDPCIS`	`AUIPC`	
Shift left logical, double word and word versions, immediate and variable	`LSL`	`SLLV, SLL`	`RLWINM`	`SLL, SLLI, SLLW, SLLWI`	`SLL`
Shift right logical, double word and word version, immediate and variables	`RSL`	`SRLV, SRL`	`RLWINM 32-i`	`SRL, SRLI, SRLW, SRLWI`	`SRL`
Shift right arithmetic, double word and word versions, immediate and variable	`RSA`	`SRAV, SRA`	`SRAW`	`SRA, SRAI, SRAW, SRAWI`	`SRA`
Compare	`CMP`	`SLT/U, SLTI/U`	`CMP(I)CLR`	`SLT/U, SLTI/U`	`SUBcc r0,...`

Figure K.10 Desktop RISC arithmetic/logical instructions equivalent to RISC-V integer ISA. MIPS also provides instructions that trap on arithmetic overflow, which are synthesized in other architectures with multiple instructions. Note that in the "Arithmetic/logical" category all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course, these are separate opcodes!)

Instruction name	**ARMv8**	**MIPS64**	**PowerPC**	**RISC-V**	**SPARC v.9**
Branch on integer compare	B.cond, CBZ, CBNZ	BEQ, BNE, B_Z (<, >, <=, >=) OR S***; BEZ	BC	BEQ, BNE, BLT, BGE, BLTU, BGEU	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	B.cond	BC1T, BC1F	BC	BEZ, BNZ	FBPfcc (<, >, <=, >=, =,...)
Jump, jump register	B, BR	J, JR	B, BCLR, BCCTR	JAL, JALR (with x0)	BA, JMPL r0,...
Call, call register	BL, BLR	JAL, JALR	BL, BLA, BCLRL, BCCTRL	JAL, JALR	CALL, JMPL
Trap	SVC, HVC, SMC	BREAK	TW, TWI	ECALL	Ticc, SIR
Return from interrupt	ERET	JR; ERET	RFI	EBREAK	DONE, RETRY, RETURN

Figure K.11 Desktop RISC control instructions equivalent to RV64G.

	ARMv8	**MIPS64**	**PowerPC**	**RISC-V**	**SPARC v.9**
Number of condition code bits (integer and FP)	16 (8 + the inverse)	none	8 × 4 both	none	2 × 4 integer, 4 × 2 FP
Basic compare instructions (integer and FP)	1 integer; 1 FP	1 integer, 1 FP	4 integer, 2 FP	2 integer; 3 FP	1 FP
Basic branch instructions (integer and FP)	1	2 integer, 1 FP	1 both	4 integer (used for FP as well)	3 integer, 1 FP
Compare register with register/constant and branch	—	=, not=	—	=, not =, >=, <	—
Compare register to zero and branch	—	=, not=, <, <=, >, >=	—	=, not=, <, <=, >, >=	=, not=, <, <=, >, >=

Figure K.12 Summary of five desktop RISC approaches to conditional branches. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using `r0` as the destination.

RISC-V and MIPS are most similar. RISC-V uses a compare and branch with a full set of arithmetic comparisons. MIPS also uses compare and branch, but the comparisons are limited to equality and tests against zero. This limited set of conditions simplifies the branch determination (since an ALU operation is not required to test the condition), at the cost of sometimes requiring the use of a set-on-less-than instruction (`SLT`, `SLTI`, `SLTU`, `SLTIU`), which compares two operands and then set the destination register to 1 if less and to 0 otherwise. Figure K.12 provides

Floating point (instruction formats)	**R-R**	**R-R**	**R-R**	**R-R**	**R-R**
Instruction name	**ARMv8**	**MIPS64**	**PowerPC**	**RISC-V**	**SPARC v.9**
Add single, double	FADD	ADD.*	FADD*	FADD.*	FADD*
Subtract single, double	FSUB	SUB.*	FSUB*	FSUB.*	FSUB*
Multiply single, double	FMUL	MUL.*	FMUL*	FMUL.*	FMUL*
Divide single, double	FDIV	DIV.*	FDIV*	FDIV.*	FDIV*
Square root single, double	FSQRT	SQRT.*	FSQRT*	FSQRT.*	FSQRT*
Multiply add; Negative multiply add: single, double	FMADD, FNMADD	MADD.* NMAD.*	FMADD*, FNMADD*	FMADD.* FNMADD.*	
Multiply subtract single, double, Negative multiply subtract: single, double	FMSUB, FNMSUB	MSUB.*, NMSUB.*	FMSUB*, FNMSUB*	FMSUB.*, FNMSUB.*	
Copy sign or negative sign double or single to another FP register	FMOV, FNEG	FMOV.*, FNEG.*	FMOV*, FNEG*	FSGNJ.*, FSGNJN.*	FMOV*, FNEG*
Replace sign bit with XOR of sign bits single double	FABS	FABS.*	FABS*	FSGNJX.*	FABS*
Maximum or minimum single, double	FMAX, FMIN	MAX.*, MIN.*		FMAX.*, FMIN.*	
Classify floating point value single double		CLASS.*		FCLASS.*	
Compare	FCMP	CMP.*	FCMP*	FCMP.*	FCMP*
Convert between FP single or double and FP single or double, OR integer single or double, signed and unsigned with rounding	FCVT	CVT, CEIL, FLOOR		FCVT	F*TO*

Figure K.13 Desktop RISC floating-point instructions equivalent to RV64G ISA with an empty entry meaning that the instruction is unavailable. ARMv8 uses the same assembly mnemonic for single and double precision; the register designator indicates the precision. "*" is used as an abbreviation for S or D. For floating point compares all conditions: equal, not equal, less than, and less-then or equal are provided. Moves operate in both directions from/to integer registers. Classify sets a register based on whether the floating point quantity is plus or minus infinity, denorm, +/ − 0, etc.). The sign-injection instructions take two operands, but are primarily used to form floating point move, negate, and absolute value, which are separate instructions in the other ISAs.

additional details on conditional branch. RISC-V floating point comparisons sets an integer register to 0 or 1, and then use conditional branches on that content.MIPS also uses separate floating-point compare, which sets a floating point register to 0 or 1, which is then tested by a floating-point conditional branch.

ARM is similar to SPARC, in that it provides four traditional condition codes that are optionally set. `CMP` subtracts one operand from the other and the difference sets the condition codes. Compare negative (`CMN`) *adds* one operand to the other, and the sum sets the condition codes. `TST` performs logical `AND` on the two operands to set all condition codes but overflow, while `TEQ` uses exclusive OR to set the first three condition codes. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations. ARMv8 added both bit-test instructions and also compare and branch against zero. Floating point compares on ARM, set the integer condition codes, which are used by the `B.cond` instruction.

As Figure K.13 shows the floating point support is similar on all five architectures.

RV64GC Core 16-bit Instructions

Figures K.14 through K.17 summarize the data transfer, ALU, and control instructions for our three embedded processors: microMIPS64, RV64GC, and Thumb-2. Since these architectures are all based on 32-bit or 64-bit versions of the full architecture, we focus our attention on the functionality implemented by the 16-bit instructions. Since floating point is optional, we do not include it. I

Instruction name	**microMIPS64 rs1;rs2/dst; offset**	**RV64GC rs1;rs2/dst; offset**	**Thumb-2 rs1;rs2/dst; offset**
Load word	8;8;4	8;8;5	8;8;5
Load double word		8;8;5	
Load word with stack pointer as base register	1;32;5	1;32;6	1;3;8
Load double word with stack pointer as base register		1;32;6	
Store word	8;8;4	8;8;5	8;8;5
Store double word		8;8;5	
Store word with stack pointer as base register	1;32;5	1;32;6	1;3;8
Store double with stack pointer as base register		1;32;6	

Figure K.14 Embedded RISC data transfer instructions equivalent to RV64GC 16-bit ISA; a blank indicates that the instruction is not a 16-bit instruction. Rather than show the instruction name, where appropriate, we show the number of registers that can the base register for the address calculation, followed by the number of registers that can be the destination for a load or the source for a store, and finally, the size of the immediate used for address calculation. For example: 8; 8; 5 for a load means that there are 8 possible base registers, 8 possible destination registers for the load, and a 5-bit offset for the address calculation. For a store, 8; 8; 5, specifies that the source of the value to store comes from one of 8 registers. Remember that Thumb-2 also has 32-bit instructions (although not the full ARMv8 set) and that RV64GC and microMIPS64 have the full set of 32-bit instructions in RV64I or MIPS64.

Instruction Name/Function	microMIPS64	RV64GC	Thumb-2
Load immediate	8;7	32;6	8;8
Load upper immediate		32;6	
add immediate	32;4	32;6	8;8;3
add immediate word (32 bits) & sign extend		32;6	
add immediate to stack pointer	1;9	1;6 (adds 16x imm.)	1;7
add immediate to stack pointer store in reg.	1;8;6	1;8;6 (adds 4x imm.)	
shift left/right logical	8;8;3 (shift amt.)	8;6(shift amt.)	8;8;5 (shift amt.)
shift right arithmetic		8;6(shift amt.)	8;8;5 (shift amt.)
AND immediate	8;8;4	8;6	8;8
move	32;32	32;32	16;16
add	8;8;8	32;32	8;8;8 16;16
AND, OR, XOR	8;8	8;8	8;8
subtract	8;8;8	8;8	8;8;8
add word, subtract word (32 bits) & sign extend		8;8	

Figure K.15 ALU instructions provided in RV64GC and the equivalents, if any, in the 16-bit instructions of microMIPS64 or Thumb-2. An entry shows the number of register sources/destinations, followed by the size of the immediate field, if it exists for that instruction. The add to stack pointer with scaled immediate instructions are used for adjusting the stack pointer and creating a pointer to a location on the stack. In Thumb, the add has two forms one with three operands from the 8-register subset (Lo) and one with two operands but any of 16-registers.

	microMIPS64	RV64GC	Thumb-2
Unconditional branch	10-bit offset	11-bit offset	11-bit offset
Unconditional branch and link		11-bit offset	11-bit offset
Unconditional branch to register w/wo link	any of 32 registers	any of 32 registers	
Compare register to zero (=/!=) and branch	8 registers; 7-bit offset	8 registers; 8-bit offset	no: but see caption

Figure K.16 Summary of three embedded RISC approaches to conditional branches. A blank indicates that the instruction does not exist. Thumb-2 uses 4 condition code bits; it provides a conditional branch that tests the 4-bit condition code and has a branch offset of 8 bits.

Function	**Definition**	**ARMv8**	**MIPS64**	**PowerPC**	**SPARC v.9**
Load/store multiple registers	Loads or stores 2 or more registers	`Load pair, store pair`		`Load store multiple (<=31 registers),`	
Cache manipulation and prefetch	Modifies status of a cache line or does a prefetch	`Prefetch`	`CACHE, PREFETCH`	`Prefetch`	`Prefetch`

Figure K.17 Data transfer instructions not found in RISC-V core but found in two or more of the five desktop architectures. SPARC requires memory accesses to be aligned, while the other architectures support unaligned access, albeit, often with major performance penalties. The other architectures do not require alignment, but may use slow mechanisms to handle unaligned accesses.MIPS provides a set of instructions to handle misaligned accesses: LDL and LDR (load double left and load double right instructions) work as a pair to load a misaligned word; the corresponding store instructions perform the inverse. The Prefetch instruction causes a cache prefetch, while CACHE provides limited user control over the cache state.

Instructions: Common Extensions beyond RV64G

Figures K.15 through K.18 list instructions not found in Figures K.9 through K.13 in the same four categories (data transfer, ALU, and control. The only significant floating point extension is the reciprocal instruction, which both MIPS64 and Power support. Instructions are put in these lists if they appear in more than one of the standard architectures. Recall that Figure K.3 on page 6 showed the address modes supported by the various instruction sets. All three processors provide more address modes than provided by RV64G. The loads and stores using these additional address modes are not shown in Figure K.17, but are effectively additional data transfer instructions. This means that ARM has 64 additional load and store instructions, while Power3 has 12, and MIPS64 and SPARVv9 each have 4.

To accelerate branches, modern processors use dynamic branch prediction (see Section 3.3). Many of these architectures in earlier versions supported delayed branches, although they have been dropped or largely eliminated in later versions

Name	**Definition**	**ARMv8**	**MIPS64**	**PowerPC**	**SPARC v.9**
Delayed branches	Delayed branches with/without cancellation		`BEQ, BNE, BGTZ, BLEZ, BCxEQZ, BCxNEZ`		`BPcc, A, FPBcc, A`
Conditional trap	Traps if a condition is true		`TEQ, TNE, TGE, TLT, TGEU, TLTU`	`TW, TD, TWI, TDI`	`Tcc`

Figure K.18 Control instructions not found in RV64G core but found in two or more of the other architectures. MIPS64 Release 6 has nondelayed and normal delayed branches, while SPARC v.9 has delayed branches with cancellation based on the static prediction.

of the architecture, typically by offering a nondelayed version, as the preferred conditional branch. The SPARC "annulling" branch is an optimized form of delayed branch that executes the instruction in the delay slot only if the branch is taken; otherwise, the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does *not* execute the following instruction.).

In contrast to the differences among the full ISAs, the 16-bit subsets of the three embedded ISAs have essentially no significant differences other than those described in the earlier figures (e.g. size of immediate fields, uses of SP or other registers, etc.).

Now that we have covered the similarities, we will focus on the unique features of each architecture. We first cover the desktop/server RISCs, ordering them by length of description of the unique features from shortest to longest, and then the embedded RISCs.

Instructions Unique to MIPS64 R6

MIPS has gone through six generations of instruction sets. Generations 1–4 mostly added instructions. Release 6 eliminated many older instructions but also provided support for nondelayed branches and misaligned data access. Figure K.19 summarizes the unique instructions in MIPS64 R6.

Instruction class	Instruction name(s)	Function
ALU	Byte align	Take a pair of registers and extract a word or double word of bytes. Used to implement unaligned byte copies.
	Align Immediate to PC	Adds the upper 16 bits of the PC to an immediate shifted left 16 bits and puts the result in a register; Used to get a PC-relative address.
	Bit swap	Reverses the bits in each byte of a register.
	No-op and link	Puts the value of PC+8 into a register
	Logical NOR	Computes the NOR of 2 registers
Control transfer	Branch and Link conditional	Compares a register to 0 and does a branch if condition is true; places the return address in the link register.
	Jump indexed, Jump and link indexed	Adds an offset and register to get new PC, w/wo link address

Figure K.19 Additional instructions provided MIPS64 R6. In addition, there are several instructions for supporting virtual machines, most are privileged.

Instructions Unique to SPARC v.9

Several features are unique to SPARC. We review the major figures and then summarize those and small differences in a figure.

Register Windows

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer. The knee of the cost-performance curve seems to be six to eight banks; programs with deeper call stacks, would need to save and restore the registers to memory.

SPARC can have between 2 and 32 windows, typically using 8 registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to "save" the caller's window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller's window of the addition operation, while the destination register is in the callee's window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of `SAVE`, bringing back the caller's window while acting as an add instruction, with the source registers from the callee's window and the destination register in the caller's window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee's final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. With the advent of multiple issue, which requires many more register ports, as will as register renaming or reorder buffers, register windows posed a larger penalty.Register windows were a feature of the original Berkeley RISC designs, and their inclusion in SPARC was inspired by those designs. Tensilica is the only other major architecture in use today employs them, and they were not included in the RISC-V ISA.

Fast Traps

SPARCv9 includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: `RETRY` (which retries the interrupted instruction) and `DONE` (which does not). To support user-level traps, the instruction `RETURN` will return from the trap in nonprivileged mode.

Support for LISP and Smalltalk

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multi-word arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least-significant bits indicate whether the operand is an integer (coded as 00), so `TADDcc` and `TSUBcc` set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure K.20 shows both types of tag support.

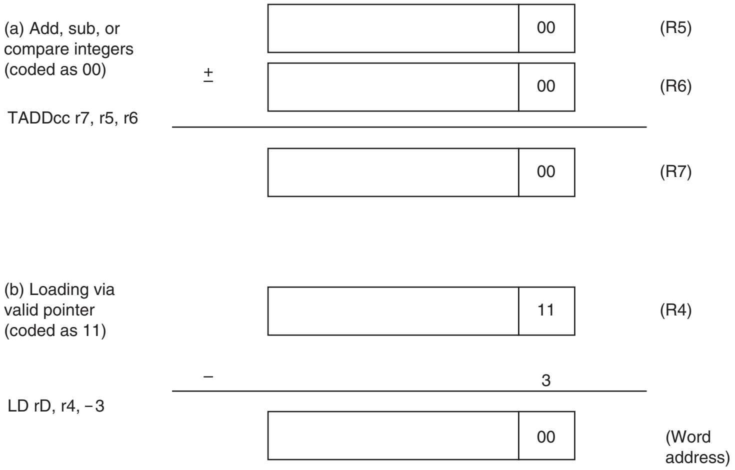

(a) Add, sub, or compare integers (coded as 00)
TADDcc r7, r5, r6
±
00
(R5)
00
(R6)
00
(R7)
(b) Loading via valid pointer (coded as 11)
LD rD, r4, −3
11
(R4)
−
3
00
(Word address)

Figure K.20 SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of −3 can be used to access the even word of a pair (CAR) and +1 can be used for the odd word of a pair (CDR).

Instruction class	**Instruction name(s)**	**Function**
Data transfer	SAVE, RESTORE	Save or restore a register window
	Nonfaulting load	Version of load instructions that do not generate faults on address exceptions; allows speculation for loads.
ALU	Tagged add, Tagged subtract, with and without trap	Perform a tagged add/subtract, set condition codes, optionally trap.
Control transfer	Retry, Return, and Done	To provide handling for traps.
Floating Point Instructions	FMOVcc	Conditional move between FP registers based on integer or FP condition codes.

Figure K.21 Additional instructions provided in SPARCv9. Although register windows are by far the most significant distinction, they do not require many instructions!

Figure K.21 summarizes the additional instructions mentioned above as well as several others.

Instructions Unique to ARM

Earlier versions of the ARM architecture (ARM v6 and v7) had a number of unusual features including conditional execution of all instructions, and making the PC a general purpose register. These features were eliminated with the arrival of ARMv8 (in both the 32-bit and 64-bit ISA). What remains, however, is much of the complexity, at least in terms of the size of the instruction set. As Figure K.3 on page 6 shows, ARM has the most addressing modes, including all those listed in the table; remember that these addressing modes add dozens of load/store instructions compared to RVG, even though they are not listed in the table that follows. As Figure K.6 on page 8 shows, ARMv8 also has by far the largest number of different instruction formats, which reflects a variety of instructions, as well as the different addressing modes, some of which are applicable to some loads and stores but not others.

Most ARMv8 ALU instructions allow the second operand to be shifted before the operation is completed. This extends the range of immediates, but operand shifting is not limited to immediates. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right. In addition, as in Power3, most ALU instructions can optionally set the condition flags. Figure K.22 includes the additional instructions, but does not enumerate all the varieties (such as optional setting of the condition flags); see the caption for more detail. While conditional execution of all instructions was eliminated, ARMv8 provides a number of conditional instructions beyond the conditional move and conditional set, mentioned earlier.

Instruction class	Instruction name(s)	Function
Data transfer	Load/Store Non-temporal pair	Loads/stores a pair of registers with an indication not to cache the data. Base + scaled offset addressing mode only.
ALU	Add Extended word/double word	Add 2 registers after left shifting the second register operand and extending it.
	Add with shift; add immediate with shift	Adds with shift of the second operand.
	Address of page	Computes the address of a page based on PC (similar to ADDUIPC, which is the same as ADR in ARMv8)
	AND, OR, XOR, XOR NOT shifted register	Logical operation on a register and a shifted register.
	Bit field clear shifted	Shift operand, invert and AND with another operand
	Conditional compare, immediate, negative, negative immediate	If condition true, then set condition flags to compare result, otherwise leave condition flags untouched.
	Conditional increment, invert, negate	If condition then set destination to increment/invert/negate of source register
	CRC	Computes a CRC checksum: byte, word, halfword, double
	Multiply add, subtract	Integer multiply-add or multiply-subtract
	Multiply negate	Negate the product of two integers; word & double word
	Move immediate or inverse	Replace 16-bits in a register with immediate, possibly shifted
	Reverse bit order	Reverses the order of bits in a register
	Signed bit field move	Move a signed bit field; sign extend to left; zero extend to right
	Unsigned divide, multiple, multiply negate, multiply-add, multiply-sub	Unsigned versions of the basic instructions
Control transfer	CBNZ, CBZ	Compare branch =/!= 0, indicating this is not a call or return.
	TBNZ, TBZ	Tests bit in a register =/!= 0, and branch.

Figure K.22 Additional instructions provided in ARMv8, the AArch64 instruction set. Unless noted the instruction is available in a word and double word format, if there is a difference. Most of the ALU instructions can optionally set the condition codes; these are not included as separate instructions here or in earlier tables.

Instructions Unique to Power3

Power3 is the result of several generations of IBM commercial RISC machines—IBM RT/PC, IBM Power1, and IBM Power2, and the PowerPC development, undertaken primarily by IBM and Motorola. First, we describe branch registers and the support for loop branches. Figure K.23 then lists the other instructions provided only in Power3.

Instruction class	**Instruction name(s)**	**Function**
Data transfer	LHBRX, LWBRX, LDBRX	Loads a halfword/word/double word but reverses the byte order.
	SHBRX, SWBRX, SDBRX	Stores a halfword/word/double word but reverses the byte order
	LDQ, STQ	Load/store quadword to a register pair.
ALU	DRAN	Generate a random number in a register
	CMPB CMPRB	Compares the individual bytes in a register and sets another register byte by byte. Compares a byte (x) against two other bytes (y and z) and sets a condition to indicate if the value of y<=x<=z.
	CRAND, CRNAND, CROR, CRNOR, CRXOR, CREQV, CORC, CRANDC	Logical operations on the condition register.
	ZCMPEQB	Compares a byte (x) against the eight bytes in another register and sets a condition to indicate if x = any of the 8 bytes
	EXTSWSL	Sign extend word and shift left
	POPCNTB, POPCNTW POPCNTD	Count number of 1s in each byte and place total in another byte. Count number of 1s in each word and place total in another word. Count number of 1s in a double word.
	PRTYD, PRTYW	Compute byte parity of the bytes in a word or double word.
	BPERMD	Permutes the bits in a double word, producing a permuted byte.
	CDTBCD, CDCBCD, ADDGCS	Instructions to convert from/to binary coded decimal (BCD) or operate on two BCD values
Control transfer	BA, BCA	Branches to an absolute address, conditionally & unconditionally
	BCCTR, BCCTRL	Conditional branch to address in the count register, w/wo linking
	BCTSAR, BCTARL	Conditional branch to address in the Branch Target Address register, w/wo linking
	CLRBHRB, MFBHRBE	Manipulate the branch history rolling buffer.
Floating Point Instructions	FRSQRTE	Computes an estimate of reciprocal of the square root,
	FTDIV, FTSQRT	Tests for divide by zero or square of negative number
	FSEL	Test register against zero and select one of two operands to move
	Decimal floating point operations	A series of 48 instructions to support decimal floating point.

Figure K.23 Additional instructions provided in Power3. Rotate instructions have two forms: one that sets a condition register and one that does not. There are a set of string instructions that load up to 32 bytes from an arbitrary address to a set of registers. These instructions will be phased out in future implementations, and hence we just mention them here.

Branch Registers: Link and Counter

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, Power3 puts the address into a special register called the *link register*. Since many procedures will return without calling another procedure, link doesn't always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, Power3 has a *count register* to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline (see Appendix C), the Power architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus, PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (`BCLR`, `BCCTR`). Figure K.23 shows the several dozen instructions that have been added; note that there is an extensive facility for decimal floating point, as well.

Instructions: Multimedia Extensions of the Desktop/Server RISCs

Support for multimedia and graphics operations developed in several phases, beginning in 1996 with Intel MMX, MIPS MDMX, and SPARC VIS. As described in Section 4.3, which we assume the reader has read, these extensions allowed a register to be treated as multiple independent small integers (8 or 16 bits long) with arithmetic and logical operations done in parallel on all the items in a register. These initial SIMD extensions, sometimes called packed SIMD, were further developed after 2000 by widening the registers, partially or totally separating them from the general purpose or floating pointer registers, and by adding support for parallel floating point operations. RISC-V has reserved an extension for such packed SIMD instructions, but the designers have opted to focus on a true vector extension for the present. The vector extension RV64V is a vector architecture, and, as Section 4.3 points out, a true vector instruction set is considerably more general, and can typically perform the operations handled by the SIMD extensions using vector operations.

Figure K.24 shows the basic structure of the SIMD extensions in ARM, MIPS, Power, and SPARC. Note the difference in how the SIMD "vector registers" are structured: repurposing the floating point, extending the floating point, or adding additional registers. Other key differences include support for FP as well as integers,

	ARMv8	**MIPS64 R6**	**Power v3.0**	**SPARCv9**
Name of ISA extension	Advanced SIMD	MIPS64 SIMD Architecture	Vector Facility	VIS
Date of Current Version	2011	2012	2015	1995
Vector registers: # x size	32 x 128 bits	32 x 128 bits	32 x 128 bits	32 x 64 bits
Use GP/FP registers or independent set	extend FP registers doubling width	extend FP registers doubling width	Independent	Same as FP registers
Integer data sizes	8, 16, 32, 64	8, 16, 32, 64	8, 16, 32, 64, 128	8,16, 32
FP data sizes	32, 64	32, 64	32	
Immediates for integer and logical operations		5 bits arithmetic 8 bits logical		

Figure K.24 Structure of the SIMD extensions intended for multimedia support. In addition to the vector facility, The last row states whether the SIMD instruction set supports immediates (e.g, add vector immediate or AND vector immediate); the entry states the size of immediates for those ISAs that support them. Note that the fact that an immediate is present is encoded in the opcode space, and could alternatively be added to the next table as additional instructions. Power 3 has an optional Vector-Scalar Extension. The Vector-Scalar Extension defines a set of vector registers that overlap the FP and normal vector registers, eliminating the need to move data back and forth to the vector registers. It also supports double precision floating point operations.

support for 128-bit integers, and provisions for immediate fields as operands in integer and logical operations. Standard load and store instructions are used for moving data from the SIMD registers to memory with special extensions to handle moving less than a full SIMD register. SPARC VIS, which was one of the earliest ISA extensions for graphics, is much more limited: only add, subtract, and multiply are included, there is no FP support, and only limited instructions for bit element operations; we include it in Figure K.24 but will not be going into more detail.

Figure K.25 shows the arithmetic instructions included in these SIMD extensions; only those appearing in at least two extensions are included. MIPS SIMD includes many other instructions, as does the Power 3 Vector-Scalar extension, which we do not cover. One frequent feature not generally found in general-purpose microprocessors is saturating operations. Saturation means that when a calculation overflows the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic. Commonly found in digital signal processors (see the next subsection), these saturating operations are helpful in routines for filtering. Another common extension are instructions for accumulating values within a single register; the dot product instruction an the maximum/minimum instructions are typical examples.

In addition to the arithmetic instructions, the most common additions are logical and bitwise operations and instructions for doing version of permutations and packing elements into the SIMD registers. These additions are summarized in Figure K.26, Lastly, all three extensions support SIMD FP operations, as summarized in Figure K.27.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
Add/subtract	16B, 8H, 4W; 2 D	16B, 8H; 4W; 2 D	16B, 8H, 4W, 2 D, Q
Saturating add/sub	16B, 8H, 4W; 2 D	16B, 8H; 4W; 2 D	16B, 8H, 4W, 2 D, Q
Absolute value of difference	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Adjacent add & subtract (pairwise)	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Average		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Dot product add, dot product subtract	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Divide: signed, unsigned	16B, 8H, 4W	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Multiply: signed, unsigned	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Multiply add, multiply subtract	16B, 8H, 4W	16B, 8H, 4W	16B, 8H, 4W; 2 D
Maximum, signed & unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Minimum, signed & unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Modulo, signed & unsigned		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Compare equal	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Compare <, <=, signed, unsigned	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q

Figure K.25 Summary of arithmetic SIMD instructions. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits), D for double word (64 bits), and Q for quad word (128 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that some instructions–such as adjacent add/subtract, or multiply–produce results that are twice the width of the inputs (e.g. multiply on 16 bytes produces 8 halfword results). Dot product is a multiply and accumulate. The SPARC VIS instructions are aimed primarily at graphics and are structured accordingly.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
Shift right/left, logical, arithmetic	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q	16B, 8H, 4W; 2 D; Q
Count leading or trailing zeros	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
and/or/xor	Q	Q	Q
Bit insert & extract	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Population count		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D; Q
Interleave even/odd, left/right		16B, 8H, 4W; 2 D	6B, 8H, 4W; 2 D
Pack even/odd		16B, 8H, 4W; 2 D	6B, 8H, 4W; 2 D
Shuffle		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D
SPLAT		16B, 8H, 4W; 2 D	16B, 8H, 4W; 2 D

Figure K.26 Summary of logical, bitwise, permute, and pack/unpack instructions, using the same format as the previous figure. When there is a single operand the instruction applies to the entire register; for logical operations there is no difference.Interleave puts together the elements (all even, odd, leftmost or rightmost) from two different registers to create one value; it can be used for unpacking. Pack moves the even or odd elements from two different registers to the leftmost and rightmost halves of the result. Shuffle creates a from two registers based on a mask that selects which source for each item. SPLAT copies a value into each item in a register.

Instruction category	ARM Advanced SIMD	MIPS SIMD	Power Vector Facility
FP add, subtract, multiply, divide	4W, 2D	4W, 2D	4W, 2D
FP multiply add/subtract	4W, 2D	4W, 2D	4W, 2D
FP maximum/minimum	4W, 2D	4W, 2D	4W, 2D
FP SQRT and 1/SQRT	4W, 2D	4W, 2D	4W, 2D
FP Compare	4W, 2D	4W, 2D	4W, 2D
FP Convert to/from integer	4W, 2D	4W, 2D	4W, 2D

Figure K.27 Summary of floating point, using the same format as the previous figure.

Instructions: Digital Signal-Processing Extensions of the Embedded RISCs

Both Thumb2 and microMIPS32 provide instructions for DSP (Digital Signal Processing) and multimedia operations. In Thumb2, these are part of the core instruction set; in microMIPS32, they are part of the DSP extension. These extensions, which are encoded as 32-bit instructions, are less extensive than the multimedia and graphics support provided in the SIMD/Vector extensions of MIPS64 or ARMv8 (AArch64). Like those more comprehensive extensions, the ones in Thumb2 and microMIPS32 also rely on packed SIMD, but they use the existing integer registers, with a small extension to allow a wide accumulator, and only operate on integer data. RISC-V has specified that the "P" extension will support packed integer SIMD using the floating point registers, but at the time of publication, the specification was not completed.

DSP operations often include linear algebra functions and operations such as convolutions; these operations produce intermediate results that will be larger than the inputs. In Thumb2, this is handled by a set of operations that produce 64-bit results using a pair of integer registers. In microMIPS32 DSP, there are 4 64-bit accumulator registers, including the Hi-Lo register, which is already exists for doing integer multiply and divide. Both architectures provide parallel arithmetic using bytes, halfwords, and words, as in the multimedia extensions in ARMv8 and MIPS64. In addition, the MIPS DSP extension handles fractional data, such data is heavily used in DSP operations. Fractional data items have a sign bit and the remaining bits are used to represent the fraction, providing a range of values from -1.0 to 0.9999 (in decimal). MIPS DSP supports two fractional data sizes Q15 and Q31 each with one sign bit and 15 or 31 bits of fraction.

Figure K.28 shows the common operations using the same notation as was used in Figure K.25. Remember that the basic 32-bit instruction set provides additional functionality, including basic arithmetic, logical, and bit manipulation.

Function	Thumb-2	microMIPS32 DSP
Add/Subtract	4B, 2H	4B, 2Q15
Add /Subtract with saturation	4B, 2H	4B, 2Q15, Q31
Add/Subtract with Exchange (exchanges halfwords in rt, then adds first halfword and subtracts second) with optional saturation	2H	
Reduce by add (sum the values)		4B
Absolute value		2Q15, Q31
Precision reduce/increase (reduces or increases the precision of a value)		2B, Q15, 2Q15, Q31
Shifts: left, right, logical & arithmetic, with optional saturation		4B, 2H
Multiply	2H	2B, 2H, 2Q15
Multiply add/subtract (to GPR or accumulator register in MIPS)	2H	2Q15
Complex multiplication step (2 multiplies and addition/subtraction)	2H	2Q15
Multiply and accumulate (by addition or subtraction)	2H	Q15, Q31
Replicate bits		B, H
Compare: =, <, <=, sets condition field		4B, 2H
Pick (use condition bits to choose bytes or halfwords from two operands)		4B, 2H
Pack choosing a halfword from each operand		H
Extract		Q63
Move from/to accumulator		DW

Figure K.28 Summary of two embedded RISC DSP operations, showing the data types for each operation. A blank indicates that the operation is not supported as a single instruction. Byte quantities are usually unsigned. Complex multiplication step implements multiplication of complex numbers where each component is a Q15 value. ARM uses its standard condition register, while MIPS adds a set of condition bits as part of the state in the DSP extension.

Concluding Remarks

This survey covers the addressing modes, instruction formats, and almost all the instructions found in 8 RISC architectures. Although the later sections concentrate on the differences, it would not be possible to cover 8 architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figures K.9 through K.13. To contrast this homogeneity, Figure K.29 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure K.1. (Since it would be impossible to write a single section in this style for those architectures, the next three sections cover the 80x86, VAX, and IBM 360/370.) In the history of computing, there has never been such widespread agreement on computer architecture as there has been since the RISC ideas emerged in the 1980s.

	IBM 360/370	**Intel 8086**	**Motorola 68000**	**DEC VAX**
Date announced	1964/1970	1978	1980	1977
Instruction size(s) (bits)	16, 32, 48	8, 16, 24, 32, 40, 48	16, 32, 48, 64, 80	8, 16, 24, 32, ... , 432
Addressing (size, model)	24 bits, flat/ 31 bits, flat	4 + 16 bits, segmented	24 bits, flat	32 bits, flat
Data aligned?	Yes 360/No 370	No	16-bit aligned	No
Data addressing modes	2/3	5	9	=14
Protection	Page	None	Optional	Page
Page size	2 KB & 4 KB	—	0.25 to 32 KB	0.5 KB
I/O	Opcode	Opcode	Memory mapped	Memory mapped
Integer registers (size, model, number)	16 GPR × 32 bits	8 dedicated data × 16 bits	8 data and 8 address × 32 bits	15 GPR × 32 bits
Separate floating-point registers	4 × 64 bits	Optional: 8 × 80 bits	Optional: 8 × 80 bits	0
Floating-point format	IBM (floating hexadecimal)	IEEE 754 single, double, extended	IEEE 754 single, double, extended	DEC

Figure K.29 Summary of four 1970s architectures. Unlike the architectures in Figure K.1, there is little agreement between these architectures in any category. (See Section K.3 for more details on the 80x86 and Section K.4 for a description of the VAX.)

K.3 The Intel 80x86

Introduction

MIPS was the vision of a single architect. The pieces of this architecture fit nicely together and the whole architecture can be described succinctly. Such is not the case of the 80x86: It is the product of several independent groups who evolved the architecture over 20 years, adding new features to the original instruction set as you might add clothing to a packed bag. Here are important 80x86 milestones:

- 1978—The Intel 8086 architecture was announced as an assembly language–compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Whereas the 8080 was a straightforward accumulator machine, the 8086 extended the architecture with additional registers. Because nearly every register has a dedicated use, the 8086 falls somewhere between an accumulator machine and a general-purpose register machine, and can fairly be called an *extended accumulator* machine.
- 1980—The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Its architects rejected extended accumulators to go with a hybrid of stacks and registers,

essentially an *extended stack* architecture: A complete stack instruction set is supplemented by a limited set of register-memory instructions.

- 1982—The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory mapping and protection model, and by adding a few instructions to round out the instruction set and to manipulate the protection model. Because it was important to run 8086 programs without change, the 80286 offered a *real addressing mode* to make the machine look just like an 8086.
- 1985—The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 2). Like the 80286, the 80386 has a mode to execute 8086 programs without change.

This history illustrates the impact of the "golden handcuffs" of compatibility on the 80x86, as the existing software base at each step was too important to jeopardize with significant architectural changes. Fortunately, the subsequent 80486 in 1989, Pentium in 1992, and P6 in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing plus a conditional move instruction.

Since 1997 Intel has added hundreds of instructions to support multimedia by operating on many narrower data types within a single clock (see Appendix A). These SIMD or vector instructions are primarily used in hand-coded libraries or drivers and rarely generated by compilers. The first extension, called MMX, appeared in 1997. It consists of 57 instructions that pack and unpack multiple bytes, 16-bit words, or 32-bit double words into 64-bit registers and performs shift, logical, and integer arithmetic on the narrow data items in parallel. It supports both saturating and nonsaturating arithmetic. MMX uses the registers comprising the floating-point stack and hence there is no new state for operating systems to save.

In 1999 Intel added another 70 instructions, labeled SSE, as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

In 2001, Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable multimedia operations, but it also gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers as found in the RISC machines. This change has boosted performance on the Pentium 4, the first microprocessor to include SSE2 instructions. At the time of announcement, a 1.5 GHz Pentium 4 was 1.24 times faster than a 1 GHz Pentium III for SPECint2000(base), but it was 1.88 times faster for SPECfp2000(base).

In 2003 a company other than Intel enhanced the IA-32 architecture this time. AMD announced a set of architectural extensions to increase the address space for 32 to 64 bits. Similar to the transition from 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to sixteen and has 16 128-bit registers to support XMM, AMD's answer to SSE2. Rather than expand the instruction set, the primary change is adding a new mode called *long mode* that redefines the execution of all IA-32 instructions with 64-bit addresses. To address the larger number of registers, it adds a new prefix to instructions. AMD64 still has a 32-bit mode that is backwards compatible to the standard Intel instruction set, allowing a more graceful transition to 64-bit addressing than the HP/Intel Itanium. Intel later followed AMD's lead, making almost identical changes so that most software can run on either 64-bit address version of the 80x86 without change.

Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other server or desktop processor in the world. Nevertheless, its checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

We start our explanation with the registers and addressing modes, move on to the integer operations, then cover the floating-point operations, and conclude with an examination of instruction encoding.

80x86 Registers and Data Addressing Modes

The evolution of the instruction set can be seen in the registers of the 80x86 (Figure K.30). Original registers are shown in black type, with the extensions of the 80386 shown in a lighter shade, a coloring scheme followed in subsequent figures. The 80386 basically extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an "E" to their name to indicate the 32-bit version. The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure K.31.

To explain the addressing modes, we need to keep in mind whether we are talking about the 16-bit mode used by both the 8086 and 80286 or the 32-bit mode available on the 80386 and its successors. The seven data memory addressing modes supported are

- Absolute
- Register indirect
- Based
- Indexed
- Based indexed with displacement
- Based with scaled indexed
- Based with scaled indexed and displacement

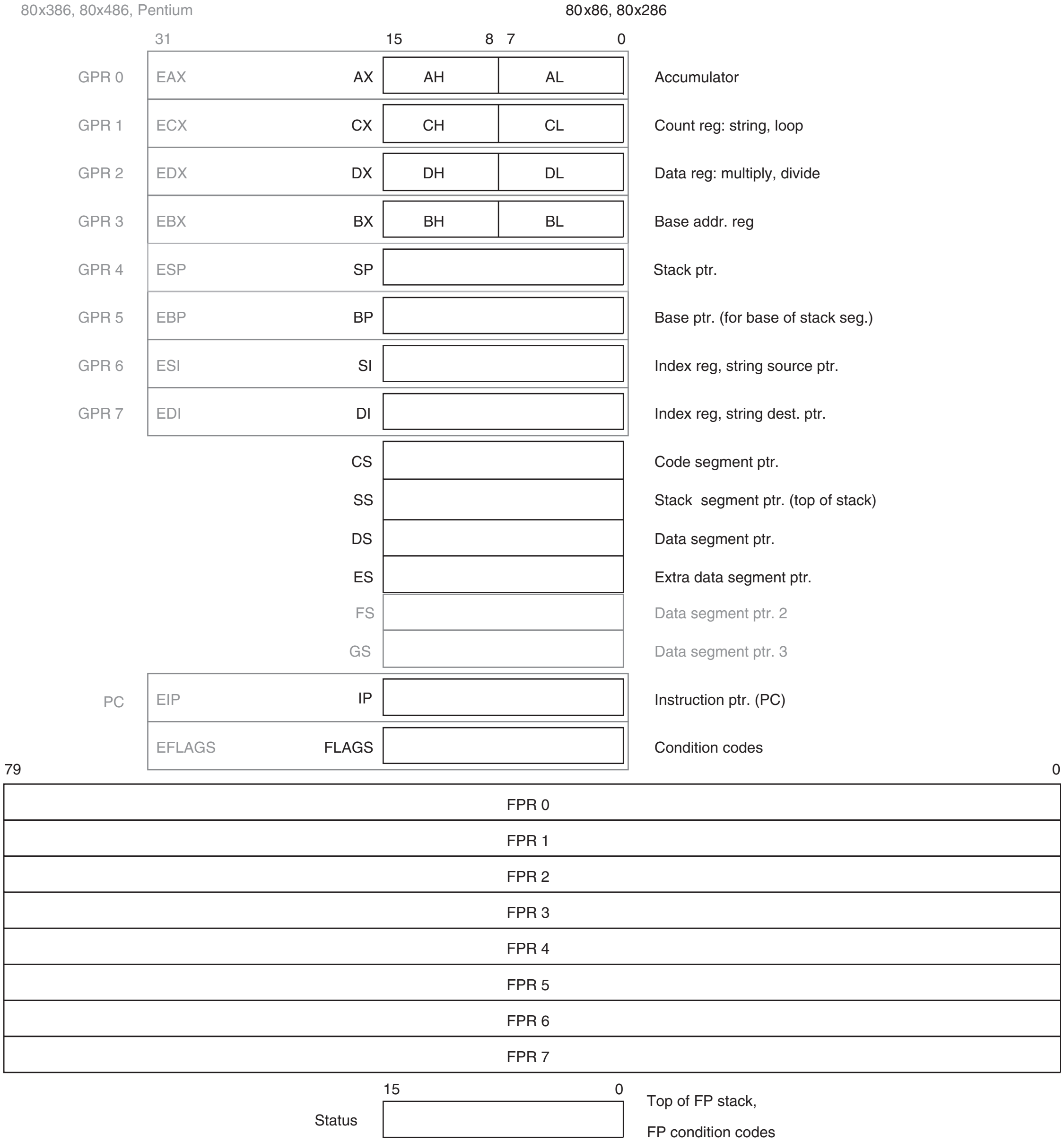

80x386, 80x486, Pentium
80x86, 80x286
31
15
8
7
0
GPR 0
EAX
AX
AH
AL
Accumulator
GPR 1
ECX
CX
CH
CL
Count reg: string, loop
GPR 2
EDX
DX
DH
DL
Data reg: multiply, divide
GPR 3
EBX
BX
BH
BL
Base addr. reg
GPR 4
ESP
SP
Stack ptr.
GPR 5
EBP
BP
Base ptr. (for base of stack seg.)
GPR 6
ESI
SI
Index reg, string source ptr.
GPR 7
EDI
DI
Index reg, string dest. ptr.
CS
Code segment ptr.
SS
Stack segment ptr. (top of stack)
DS
Data segment ptr.
ES
Extra data segment ptr.
FS
Data segment ptr. 2
GS
Data segment ptr. 3
PC
EIP
IP
Instruction ptr. (PC)
EFLAGS
FLAGS
Condition codes
79
0
FPR 0
FPR 1
FPR 2
FPR 3
FPR 4
FPR 5
FPR 6
FPR 7
15
0
Status
Top of FP stack,
FP condition codes

Figure K.30 The 80x86 has evolved over time, and so has its register set. The original set is shown in black and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers. The floating-point registers on the bottom are 80 bits wide, and although they look like regular registers they are not. They implement a stack, with the top of stack pointed to by the status register. One operand must be the top of stack, and the other can be any of the other seven registers below the top of stack.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Figure K.31 Instruction types for the arithmetic, logical, and data transfer instructions. The 80x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure K.30 (not IP or FLAGS).

Displacements can be 8 or 32 bits in 32-bit mode, and 8 or 16 bits in 16-bit mode. If we count the size of the address as a separate addressing mode, the total is 11 addressing modes.

Although a memory operand can use any addressing mode, there are restrictions on what registers can be used in a mode. The section "80x86 Instruction Encoding" on page K-11 gives the full set of restrictions on registers, but the following description of addressing modes gives the basic register options:

- *Absolute*—With 16-bit or 32-bit displacement, depending on the mode.
- *Register indirect*—`BX`, `SI`, `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode.
- *Based mode with 8-bit or 16-bit/32-bit displacement*—`BP`, `BX`, `SI`, and `DI` in 16-bit mode and `EAX`, `ECX`, `EDX`, `EBX`, `ESI`, and `EDI` in 32-bit mode. The displacement is either 8 bits or the size of the address mode: 16 or 32 bits. (Intel gives two different names to this single addressing mode, *based* and *indexed*, but they are essentially identical and we combine them. This book uses indexed addressing to mean something different, explained next.)
- *Indexed*—The address is the sum of two registers. The allowable combinations are `BX+SI`, `BX+DI`, `BP+SI`, and `BP+DI`. This mode is called *based indexed* on the 8086. (The 32-bit mode uses a different addressing mode to get the same effect.)
- *Based indexed with 8- or 16-bit displacement*—The address is the sum of displacement and contents of two registers. The same restrictions on registers apply as in indexed mode.
- *Base plus scaled indexed*—This addressing mode and the next were added in the 80386 and are only available in 32-bit mode. The address calculation is

$$\text{Base register} + 2^{\text{Scale}} \times \text{Index} \times \text{register}$$

where *Scale* has the value 0, 1, 2, or 3; *Index register* can be any of the eight 32-bit general registers except `ESP`; and *Base register* can be any of the eight 32-bit general registers.

- *Base plus scaled index with 8- or 32-bit displacement*—The address is the sum of the displacement and the address calculated by the scaled mode immediately above. The same restrictions on registers apply.

The 80x86 uses Little Endian addressing.

Ideally, we would refer discussion of 80x86 logical and physical addresses to Chapter 2, but the segmented address space prevents us from hiding that information. Figure K.32 shows the memory mapping options on the generations of 80x86 machines; Chapter 2 describes the segmented protection scheme in greater detail.

The assembly language programmer clearly must specify which segment register should be used with an address, no matter which address mode is used. To save space in the instructions, segment registers are selected automatically depending on which address register is used. The rules are simple: References to instructions (IP) use the code segment register (CS), references to the stack (BP or SP) use the stack segment register (SS), and the default segment register for the other registers is the data segment register (DS). The next section explains how they can be overridden.

80x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (called *word*) data types. The data type distinctions apply to register operations as well as memory accesses. The 80386 adds 32-bit addresses and data, called *double words.* Almost every operation works on both 8-bit data and one longer data size. That size is determined by the mode and is either 16 or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80x86 architects provide a convenient way to specify each version without expanding code size significantly. They decided that most programs would be dominated by either 16- or 32-bit data, and so it made sense to be able to set a default large size. This default size is set by a bit in the code segment register. To override the default size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus so as to perform a semaphore (see Chapter 5), or repeat the following instruction until CX counts down to zero. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.

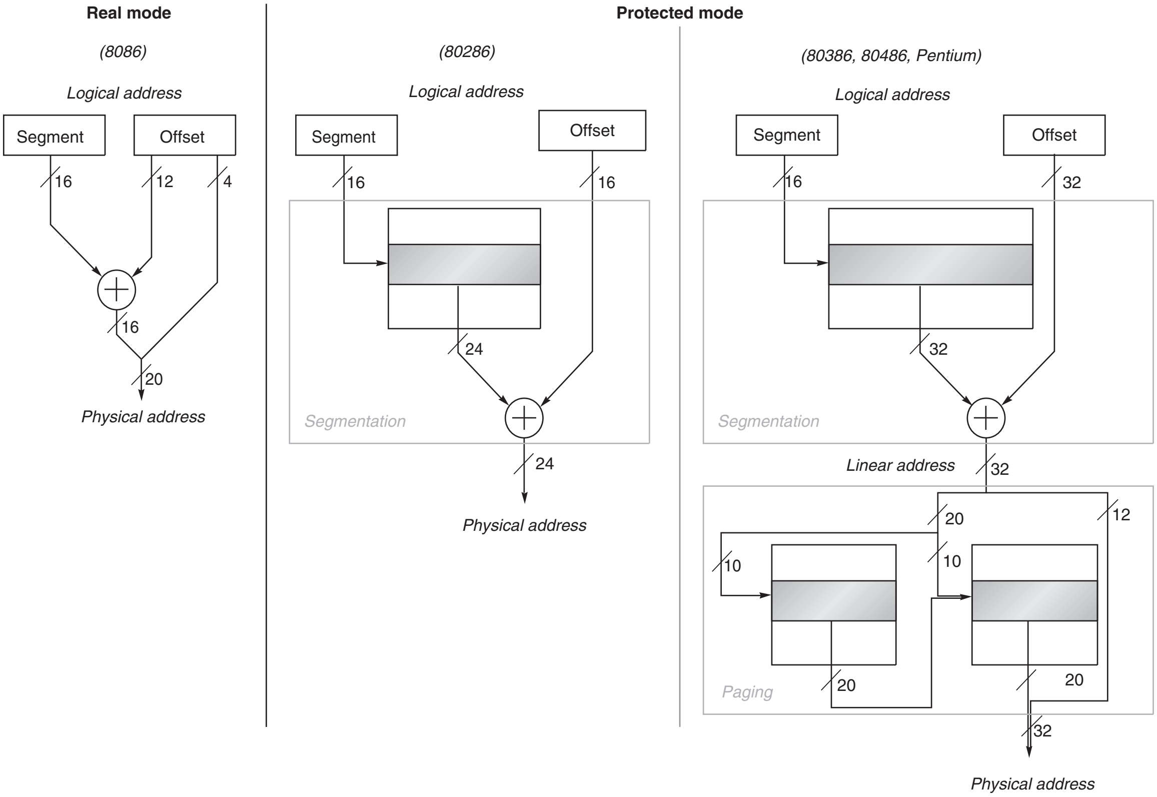

Real mode
(8086)
Logical address
Segment
Offset
16
12
4
16
20
Physical address
Protected mode
(80286)
Logical address
Segment
Offset
16
16
24
Segmentation
24
Physical address
(80386, 80486, Pentium)
Logical address
Segment
Offset
16
32
32
Segmentation
Linear address
32
20
12
10
10
20
20
Paging
32
Physical address

Figure K.32 The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (center) used the contents of the segment register to select a segment descriptor, which includes a 24-bit base address among other items. It is added to the 16-bit offset to form the 24-bit physical address. The 80386 and successors (right) expand this base address in the segment descriptor to 32 bits and also add an optional paging layer below segmentation. A 32-bit linear address is first formed from the segment and offset, and then this address is divided into two 10-bit fields and a 12-bit page offset. The first 10-bit field selects the entry in the first-level page table, and then this entry is used in combination with the second 10-bit field to access the second-level page table to select the upper 20 bits of the physical address. Prepending this 20-bit address to the final 12-bit field gives the 32-bit physical address. Paging can be turned off, redefining the 32-bit linear address as the physical address. Note that a "flat" 80x86 address space comes simply by loading the same value in all the segment registers; that is, it doesn't matter which segment register is selected.

The 80x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including logical operations, test, shifts, and integer and decimal arithmetic operations
3. Control flow, including conditional branches and unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

Instruction	Function
JE name	if equal(CC) {IP←name};IP-128≤name≤IP+128
JMP name	IP← name
CALLF name, seg	SP←SP-2;M[SS:SP]←IP+5;SP←SP-2; M[SS:SP]←CS;IP← name;CS←seg;
	MOVW BX,[DI+45] BX←$_{16}$M[DS:DI+45]
PUSH SI	SP←SP-2;M[SS:SP]←SI
POP DI	DI←M[SS:SP];SP←SP+2
ADD AX,#6765	AX←AX+6765
SHL BX,1	BX←BX$_{1..15}$## 0
TEST DX,#42	Set CC flags with DX & 42
MOVSB	M[ES:DI]←$_8$M[DS:SI];DI←DI+1;SI←SI+1

Figure K.33 Some typical 80x86 instructions and their functions. A list of frequent operations appears in Figure K.34. We use the abbreviation SR:X to indicate the formation of an address with segment register SR and offset X. This effective address corresponding to SR:X is (SR<<4)+X. The CALLF saves the IP of the next instruction and the current CS on the stack.

Figure K.33 shows some typical 80x86 instructions and their functions.

The data transfer, arithmetic, and logic instructions are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location.

Control flow instructions must be able to address destinations in another segment. This is handled by having two types of control flow instructions: "near" for intrasegment (within a segment) and "far" for intersegment (between segments) transfers. In far jumps, which must be unconditional, two 16-bit quantities follow the opcode in 16-bit mode. One of these is used as the instruction pointer, while the other is loaded into CS and becomes the new code segment. In 32-bit mode the first field is expanded to 32 bits to match the 32-bit program counter (EIP).

Calls and returns work similarly—a far call pushes the return instruction pointer and return segment on the stack and loads both the instruction pointer and the code segment. A far return pops both the instruction pointer and the code segment from the stack. Programmers or compiler writers must be sure to always use the same type of call *and* return for a procedure—a near return does not work with a far call, and *vice versa*.

String instructions are part of the 8080 ancestry of the 80x86 and are not commonly executed in most programs.

Figure K.34 lists some of the integer 80x86 instructions. Many of the instructions are available in both byte and word formats.

Instruction	Meaning
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to IP + 8-bit offset; JNE (for JNZ) and JE (for JZ) are alternative names
JMP, JMPF	Unconditional jump—8- or 16-bit offset intrasegment (near) and intersegment (far) versions
CALL, CALLF	Subroutine call—16-bit offset; return address pushed; near and far versions
RET, RETF	Pops return address from stack and jumps to it; near and far versions
LOOP	Loop branch—decrement CX; jump to IP + 8-bit displacement if CX ¦ 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between two registers or between register and memory
PUSH	Push source operand on stack
POP	Pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic/logical	**Arithmetic and logical operations using the data registers and memory**
ADD	Add source to destination; register-memory format
SUB	Subtract source from destination; register-memory format
CMP	Compare source and destination; register-memory format
SHL	Shift left
SHR	Shift logical right
RCR	Rotate right with carry as fill
CBW	Convert byte in AL to word in AX
TEST	Logical AND of source and destination sets flags
INC	Increment destination; register-memory format
DEC	Decrement destination; register-memory format
OR	Logical OR; register-memory format
XOR	Exclusive OR; register-memory format
String instructions	**Move between string operands; length given by a repeat prefix**
MOVS	Copies from string source to destination; may be repeated
LODS	Loads a byte or word of a string into the A register

Figure K.34 Some typical operations on the 80x86. Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

80x86 Floating-Point Operations

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the top two elements of the stacks, and stores can pop elements off the stack, just as the stack example in Figure A.31 on page A-4 suggests.

Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers below the top of the stack.

This hybrid is still a restricted register-memory model, however, in that loads always move data to the top of the stack while incrementing the top of stack pointer and stores can only move the top of stack to memory. Intel uses the notation `ST` to indicate the top of stack, and `ST(i)` to represent the *i*th register below the top of stack.

One novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. Memory data can be 32-bit (single-precision) or 64-bit (double-precision) floating-point numbers, called *real* by Intel. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to reals, and *vice versa*, for integer loads and stores.

The 80x86 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer CPU so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure K.35 shows some of the 60 floating-point operations. We use the curly brackets `{}` to show optional variations of the basic operations: `{I}` means there is an integer version of the instruction, `{P}` means this variation will pop one operand off the stack after the operation, and `{R}` means reverse the sense of the operands in this operation.

Not all combinations are provided. Hence,

```
F{I}SUB{R}{P}
```

represents these instructions found in the 80x86:

```
FSUB
FISUB
FSUBR
FISUBR
FSUBP
FSUBRP
```

Data transfer	Arithmetic	Compare	Transcendental
`F{I}LD mem/ST(i)`	`F{I}ADD{P}mem/ST(i)`	`F{I}COM{P}{P}`	`FPATAN`
`F{I}ST{P} mem/ST(i)`	`F{I}SUB{R}{P}mem/ST(i)`	`F{I}UCOM{P}{P}`	`F2XM1`
`FLDPI`	`F{I}MUL{P}mem/ST(i)`	`FSTSW AX/mem`	`FCOS`
`FLD1`	`F{I}DIV{R}{P}mem/ST(i)`		`FPTAN`
`FLDZ`	`FSQRT`		`FPREM`
	`FABS`		`FSIN`
	`FRNDINT`		`FYL2X`

Figure K.35 The floating-point instructions of the 80x86. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations.

There are no pop or reverse pop versions of the integer subtract instructions.

Note that we get even more combinations when including the operand modes for these operations. The floating-point add has these options, ignoring the integer and pop versions of the instruction:

`FADD`	Both operands are in the in stack, and the result replaces the top of stack.
`FADD ST(i)`	One source operand is *i*th register below the top of stack, and the result replaces the top of stack.
`FADD ST(i),ST`	One source operand is the top of stack, and the result replaces *i*th register below the top of stack.
`FADD mem32`	One source operand is a 32-bit location in memory, and the result replaces the top of stack.
`FADD mem64`	One source operand is a 64-bit location in memory, and the result replaces the top of stack.

As mentioned earlier SSE2 presents a model of IEEE floating-point registers.

80x86 Instruction Encoding

Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats. Instructions may vary from 1 byte, when there are no operands, to up to 6 bytes, when the instruction contains a 16-bit immediate and uses 16-bit displacement addressing. Prefix instructions increase 8086 instruction length beyond the obvious sizes.

The 80386 additions expand the instruction size even further, as Figure K.36 shows. Both the displacement and immediate fields can be 32 bits long, two more prefixes are possible, the opcode can be 16 bits long, and the scaled index mode specifier adds another 8 bits. The maximum possible 80386 instruction is 17 bytes long.

Figure K.37 shows the instruction format for several of the example instructions in Figure K.33. The opcode byte usually contains a bit saying whether the operand is a byte wide or the larger size, 16 bits or 32 bits depending on the mode. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form `register` ←`register op immediate`. Other instructions use a "postbyte" or extra opcode byte, labeled "mod, reg, r/m" in Figure K.36, which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The based with scaled index uses a second postbyte, labeled "sc, index, base" in Figure K.36.

The floating-point instructions are encoded in the escape opcode of the 8086 and the postbyte address specifier. The memory operations reserve 2 bits to decide

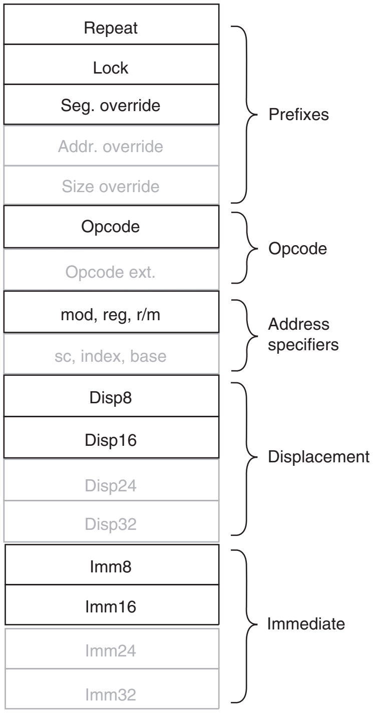
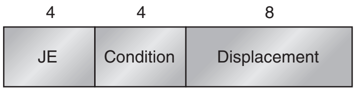
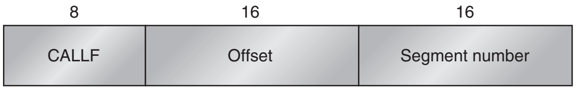

Repeat
Lock
Seg. override
Addr. override
Size override
Prefixes
Opcode
Opcode ext.
Opcode
mod, reg, r/m
sc, index, base
Address specifiers
Disp8
Disp16
Disp24
Disp32
Displacement
Imm8
Imm16
Imm24
Imm32
Immediate

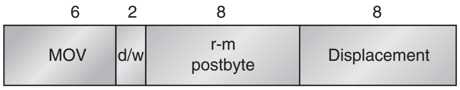
Figure K.36 The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.


4
4
8
JE
Condition
Displacement

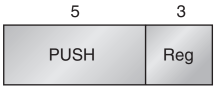
a. JE PC + displacement


8
16
16
CALLF
Offset
Segment number

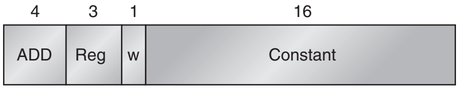
b. CALLF


6
2
8
8
MOV
d/w
r-m postbyte
Displacement

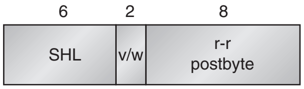
c. MOV BX, [DI + 45]



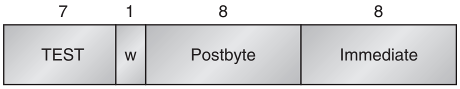
d. PUSH SI


4
3
1
16
ADD
Reg
w
Constant

e. ADD AX, #6765


6
2
8
SHL
v/w
r-r postbyte

f. SHL BX, 1


7
1
8
8
TEST
w
Postbyte
Immediate

g. TEST DX, #42

Figure K.37 Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure K.38. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a word. Fields of the form v/w or d/w are a d-field or v-field followed by the w-field. The d-field in `MOV` is used in instructions that may move to or from memory and shows the direction of the move. The field v in the `SHL` instruction indicates a variable-length shift; variable-length shifts use a register to hold the shift count. The `ADD` instruction shows a typical optimized short encoding usable only when the first operand is `AX`. Overall instructions may vary from 1 to 6 bytes in length.

whether the operand is a 32- or 64-bit real or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Alas, you cannot separate the restrictions on registers from the encoding of the addressing modes in the 80x86. Hence, Figures K.38 and K.39 show the encoding of the two postbyte address specifiers for both 16- and 32-bit mode.

		w = 1			mod = 0		mod = 1		mod = 2		
reg	**w = 0**	**16b**	**32b**	**r/m**	**16b**	**32b**	**16b**	**32b**	**16b**	**32b**	**mod = 3**
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod= 0*	*mod= 0*	*mod= 0*	*mod= 0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp 8*	*+ disp 8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	=(si)b	SI+disp16	(sib)+disp8	SI+disp8	(sib)+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Figure K.38 The encoding of the first address specifier of the 80x86, *mod, reg, r/m*. The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16- or 32-bit mode. The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16- or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod ¦3 (sib) means use the scaled index mode shown in Figure K.39. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

Index		**Base**
0	EAX	EAX
1	ECX	ECX
2	EDX	EDX
3	EBX	EBX
4	No index	ESP
5	EBP	If mod = 0, disp32 If mod ¦ 0, EBP
6	ESI	ESI
7	EDI	EDI

Figure K.39 Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the (sib) notation in Figure K.38. Note that this mode expands the list of registers to be used in other modes: Register indirect using ESP comes from Scale = 0, Index = 4, and Base = 4, and base displacement with EBP comes from Scale = 0, Index = 5, and mod = 0. The two-bit scale field is used in this formula of the effective address: Base register + 2^{Scale} × Index register.

Putting It All Together: Measurements of Instruction Set Usage

In this section, we present detailed measurements for the 80x86 and then compare the measurements to MIPS for the same programs. To facilitate comparisons among dynamic instruction set measurements, we use a subset of the SPEC92 programs. The 80x86 results were taken in 1994 using the Sun Solaris FORTRAN and C compilers V2.0 and executed in 32-bit mode. These compilers were comparable in quality to the compilers used for MIPS.

Remember that these measurements depend on the benchmarks chosen and the compiler technology used. Although we feel that the measurements in this section are reasonably indicative of the usage of these architectures, other programs may behave differently from any of the benchmarks here, and different compilers may yield different results. In doing a real instruction set study, the architect would want to have a much larger set of benchmarks, spanning as wide an application range as possible, and consider the operating system and its usage of the instruction set. Single-user benchmarks like those measured here do not necessarily behave in the same fashion as the operating system.

We start with an evaluation of the features of the 80x86 in isolation, and later compare instruction counts with those of DLX.

Measurements of 80x86 Operand Addressing

We start with addressing modes. Figure K.40 shows the distribution of the operand types in the 80x86. These measurements cover the "second" operand of the operation; for example,

```
mov EAX, [45]
```

counts as a single memory operand. If the types of the first operand were counted, the percentage of register usage would increase by about a factor of 1.5.

The 80x86 memory operands are divided into their respective addressing modes in Figure K.41. Probably the biggest surprise is the popularity of the

	Integer average	FP average
Register	45%	22%
Immediate	16%	6%
Memory	39%	72%

Figure K.40 Operand type distribution for the average of five SPECint92 programs (compress, eqntott, espresso, gcc, li) and the average of five SPECfp92 programs (doduc, ear, hydro2d, mdljdp2, su2cor).

Addressing mode	Integer average	FP average
Register indirect	13%	3%
Base + 8-bit disp.	31%	15%
Base + 32-bit disp.	9%	25%
Indexed	0%	0%
Based indexed + 8-bit disp.	0%	0%
Based indexed + 32-bit disp.	0%	1%
Base + scaled indexed	22%	7%
Base + scaled indexed + 8-bit disp.	0%	8%
Base + scaled indexed + 32-bit disp.	4%	4%
32-bit direct	20%	37%

Figure K.41 Operand addressing mode distribution by program. This chart does not include addressing modes used by branches or control instructions.

addressing modes added by the 80386, the last four rows of the figure. They account for about half of all the memory accesses. Another surprise is the popularity of direct addressing. On most other machines, the equivalent of the direct addressing mode is rare. Perhaps the segmented address space of the 80x86 makes direct addressing more useful, since the address is relative to a base address from the segment register.

These addressing modes largely determine the size of the Intel instructions. Figure K.42 shows the distribution of instruction sizes. The average number of bytes per instruction for integer programs is 2.8, with a standard deviation of 1.5, and 4.1 with a standard deviation of 1.9 for floating-point programs. The difference in length arises partly from the differences in the addressing modes: Integer programs rely more on the shorter register indirect and 8-bit displacement addressing modes, while floating-point programs more frequently use the 80386 addressing modes with the longer 32-bit displacements.

Given that the floating-point instructions have aspects of both stacks and registers, how are they used? Figure K.43 shows that, at least for the compilers used in this measurement, the stack model of execution is rarely followed. (See Section L.3 for a historical explanation of this observation.)

Finally, Figures K.44 and K.45 show the instruction mixes for 10 SPEC92 programs.

Comparative Operation Measurements

Figures K.46 and K.47 show the number of instructions executed for each of the 10 programs on the 80x86 and the ratio of instruction execution compared with that

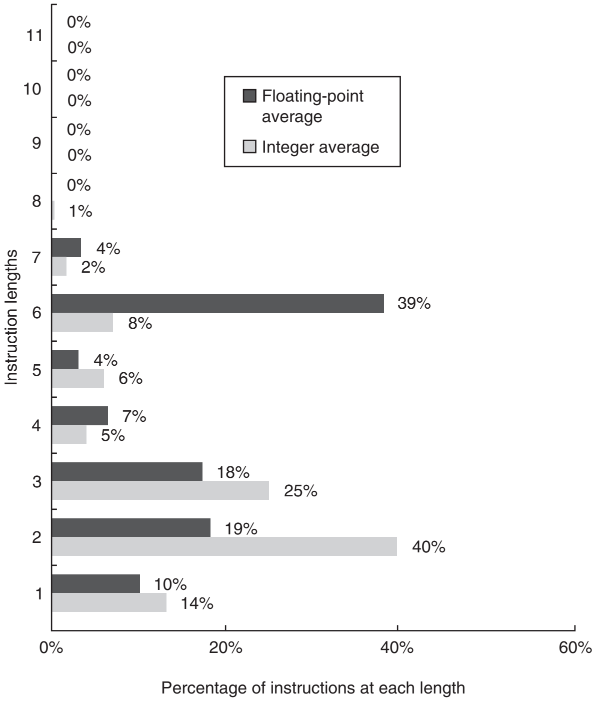


Figure K.42 Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp92 programs, all running in 32-bit mode.

Option	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Stack (2nd operand ST (1))	1.1%	0.0%	0.0%	0.2%	0.6%	0.4%
Register (2nd operand ST(i), i > 1)	17.3%	63.4%	14.2%	7.1%	30.7%	26.5%
Memory	81.6%	36.6%	85.8%	92.7%	68.7%	73.1%

Figure K.43 The percentage of instructions for the floating-point operations (add, sub, mul, div) that use each of the three options for specifying a floating-point operand on the 80x86. The three options are (1) the strict stack model of implicit operands on the stack, (2) register version naming an explicit operand that is not one of the top two elements of the stack, and (3) memory operand.

for DLX: Numbers less than 1.0 mean that the 80x86 executes fewer instructions than DLX. The instruction count is surprisingly close to DLX for many integer programs, as you would expect a load-store instruction set architecture like DLX to execute more instructions than a register-memory architecture like the 80x86. The floating-point programs always have higher counts for the 80x86,

Instruction	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Load	8.9%	6.5%	18.0%	27.6%	27.6%	20%
Store	12.4%	3.1%	11.5%	7.8%	7.8%	8%
Add	5.4%	6.6%	14.6%	8.8%	8.8%	10%
Sub	1.0%	2.4%	3.3%	2.4%	2.4%	3%
Mul						0%
Div						0%
Compare	1.8%	5.1%	0.8%	1.0%	1.0%	2%
Mov reg-reg	3.2%	0.1%	1.8%	2.3%	2.3%	2%
Load imm	0.4%	1.5%				0%
Cond. branch	5.4%	8.2%	5.1%	2.7%	2.7%	5%
Uncond branch	0.8%	0.4%	1.3%	0.3%	0.3%	1%
Call	0.5%	1.6%		0.1%	0.1%	0%
Return, jmp indirect	0.5%	1.6%		0.1%	0.1%	0%
Shift	1.1%		4.5%	2.5%	2.5%	2%
AND	0.8%	0.8%	0.7%	1.3%	1.3%	1%
OR	0.1%			0.1%	0.1%	0%
Other (XOR, not, . . .)						0%
Load FP	14.1%	22.5%	9.1%	12.6%	12.6%	14%
Store FP	8.6%	11.4%	4.1%	6.6%	6.6%	7%
Add FP	5.8%	6.1%	1.4%	6.6%	6.6%	5%
Sub FP	2.2%	2.7%	3.1%	2.9%	2.9%	3%
Mul FP	8.9%	8.0%	4.1%	12.0%	12.0%	9%
Div FP	2.1%		0.8%	0.2%	0.2%	0%
Compare FP	9.4%	6.9%	10.8%	0.5%	0.5%	5%
Mov reg-reg FP	2.5%	0.8%	0.3%	0.8%	0.8%	1%
Other (abs, sqrt, . . .)	3.9%	3.8%	4.1%	0.8%	0.8%	2%

Figure K.44 80x86 instruction mix for five SPECfp92 programs.

presumably due to the lack of floating-point registers and the use of a stack architecture.

Another question is the total amount of data traffic for the 80x86 versus DLX, since the 80x86 can specify memory operands as part of operations while DLX can only access via loads and stores. Figures K.46 and K.47 also show the data reads, data writes, and data read-modify-writes for these 10 programs. The total

Instruction	compress	eqntott	espresso	gcc (cc1)	li	Int. average
Load	20.8%	18.5%	21.9%	24.9%	23.3%	22%
Store	13.8%	3.2%	8.3%	16.6%	18.7%	12%
Add	10.3%	8.8%	8.15%	7.6%	6.1%	8%
Sub	7.0%	10.6%	3.5%	2.9%	3.6%	5%
Mul				0.1%		0%
Div						0%
Compare	8.2%	27.7%	15.3%	13.5%	7.7%	16%
Mov reg-reg	7.9%	0.6%	5.0%	4.2%	7.8%	4%
Load imm	0.5%	0.2%	0.6%	0.4%		0%
Cond. branch	15.5%	28.6%	18.9%	17.4%	15.4%	20%
Uncond. branch	1.2%	0.2%	0.9%	2.2%	2.2%	1%
Call	0.5%	0.4%	0.7%	1.5%	3.2%	1%
Return, jmp indirect	0.5%	0.4%	0.7%	1.5%	3.2%	1%
Shift	3.8%		2.5%	1.7%		1%
AND	8.4%	1.0%	8.7%	4.5%	8.4%	6%
OR	0.6%		2.7%	0.4%	0.4%	1%
Other (XOR, not, . . .)	0.9%		2.2%	0.1%		1%
Load FP						0%
Store FP						0%
Add FP						0%
Sub FP						0%
Mul FP						0%
Div FP						0%
Compare FP						0%
Mov reg-reg FP						0%
Other (abs, sqrt, . . .)						0%

Figure K.45 **80x86 instruction mix for five SPECint92 programs.**

accesses ratio to DLX of each memory access type is shown in the bottom rows, with the read-modify-write counting as one read and one write. The 80x86 performs about two to four times as many data accesses as DLX for floating-point programs, and 1.25 times as many for integer programs. Finally, Figure K.48 shows the percentage of instructions in each category for 80x86 and DLX.

	compress	eqntott	espresso	gcc (cc1)	li	Int. avg.
Instructions executed on 80x86 (millions)	2226	1203	2216	3770	5020	
Instructions executed ratio to DLX	0.61	1.74	0.85	0.96	0.98	1.03
Data reads on 80x86 (millions)	589	229	622	1079	1459	
Data writes on 80x86 (millions)	311	39	191	661	981	
Data read-modify-writes on 80x86 (millions)	26	1	129	48	48	
Total data reads on 80x86 (millions)	615	230	751	1127	1507	
Data read ratio to DLX	0.85	1.09	1.38	1.25	0.94	1.10
Total data writes on 80x86 (millions)	338	40	319	709	1029	
Data write ratio to DLX	1.67	9.26	2.39	1.25	1.20	3.15
Total data accesses on 80x86 (millions)	953	269	1070	1836	2536	
Data access ratio to DLX	1.03	1.25	1.58	1.25	1.03	1.23

Figure K.46 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.

	doduc	ear	hydro2d	mdljdp2	su2cor	FP average
Instructions executed on 80x86 (millions)	1223	15,220	13,342	6197	6197	
Instructions executed ratio to DLX	1.19	1.19	2.53	2.09	1.62	1.73
Data reads on 80x86 (millions)	515	6007	5501	3696	3643	
Data writes on 80x86 (millions)	260	2205	2085	892	892	
Data read-modify-writes on 80x86 (millions)	1	0	189	124	124	
Total data reads on 80x86 (millions)	517	6007	5690	3820	3767	
Data read ratio to DLX	2.04	2.36	4.48	4.77	3.91	3.51
Total data writes on 80x86 (millions)	261	2205	2274	1015	1015	
Data write ratio to DLX	3.68	33.25	38.74	16.74	9.35	20.35
Total data accesses on 80x86 (millions)	778	8212	7965	4835	4782	
Data access ratio to DLX	2.40	3.14	5.99	5.73	4.47	4.35

Figure K.47 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.

Concluding Remarks

Beauty is in the eye of the beholder.

Old Adage

As we have seen, "orthogonal" is not a term found in the Intel architectural dictionary. To fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes the encoding of the instructions.

	Integer average		FP average	
Category	**x86**	**DLX**	**x86**	**DLX**
Total data transfer	34%	36%	28%	2%
Total integer arithmetic	34%	31%	16%	12%
Total control	24%	20%	6%	10%
Total logical	8%	13%	3%	2%
Total FP data transfer	0%	0%	22%	33%
Total FP arithmetic	0%	0%	25%	41%

Figure K.48 Percentage of instructions executed by category for 80x86 and DLX for the averages of five SPECint92 and SPECfp92 programs of Figures K.46 and K.47.

Some argue that the inelegance of the 80x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time some features may be seen as undesirable. The awkwardness of the 80x86 began at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions of the 8087, 80286, and 80386.

A counterexample is the IBM 360/370 architecture, which is much older than the 80x86. It dominates the mainframe market just as the 80x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the 80x86 more than 30 years after its first implementation.

For better or worse, Intel had a 16-bit microprocessor years before its competitors' more elegant architectures, and this head start led to the selection of the 8086 as the CPU for the IBM PC. What it lacks in style is made up in quantity, making the 80x86 beautiful from the right perspective.

The saving grace of the 80x86 is that its architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. High floating-point performance is a larger challenge in this architecture.

K.4 The VAX Architecture

VAX: the most successful minicomputer design in industry history . . . the VAX was probably the hacker's favorite machine . . . . Especially noted for its large, assembler-programmer-friendly instruction set—an asset that became a liability after the RISC revolution.

Eric Raymond
The New Hacker's Dictionary (1991)

Introduction

To enhance your understanding of instruction set architectures, we chose the VAX as the representative *Complex Instruction Set Computer* (CISC) because it is so different from MIPS and yet still easy to understand. By seeing two such divergent styles, we are confident that you will be able to learn other instruction sets on your own.

At the time the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages in order to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions. As VAX architect William Strecker said ("VAX-11/780—A Virtual Address Extension to the PDP-11 Family," *AFIPS Proc.,* National Computer Conference, 1978):

> A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal: 1) A very regular and consistent treatment of operators . . . . 2) An avoidance of instructions unlikely to be generated by a compiler . . . . 3) Inclusions of several forms of common operators . . . . 4) Replacement of common instruction sequences with single instructions . . . . Examples include procedure calling, multiway branching, loop control, and array subscript calculation.

Recall that DRAMs of the mid-1970s contained less than 1/1000th the capacity of today's DRAMs, so code space was also critical. Hence, another prevailing philosophy was to minimize code size, which is de-emphasized in fixed-length instruction sets like MIPS. For example, MIPS address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Whole books have been written just about the VAX, so this VAX extension cannot be exhaustive. Hence, the following sections describe only a few of its addressing modes and instructions. To show the VAX instructions in action, later sections show VAX assembly code for two C procedures. The general style will be to contrast these instructions with the MIPS code that you are already familiar with.

The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to the powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. The MIPS goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

VAX Operands and Addressing Modes

The VAX is a 32-bit architecture, with 32-bit-wide addresses and 32-bit-wide registers. Yet, the VAX supports many other data sizes and types, as Figure K.49 shows. Unfortunately, VAX uses the name "word" to refer to 16-bit quantities; in this text, a word means 32 bits. Figure K.49 shows the conversion between

Bits	Data type	MIPS name	VAX name
8	Integer	Byte	Byte
16	Integer	Half word	Word
32	Integer	Word	Long word
32	Floating point	Single precision	F_floating
64	Integer	Double word	Quad word
64	Floating point	Double precision	D_floating or G_floating
8n	Character string	Character	Character

Figure K.49 VAX data types, their lengths, and names. The first letter of the VAX type (b, w, l, f, q, d, g, c) is often used to complete an instruction name. Examples of move instructions include `movb`, `movw`, `movl`, `movf`, `movq`, `movd`, `movg`, and `movc3`. Each move instruction transfers an operand of the data type indicated by the letter following `mov`.

the MIPS data type names and the VAX names. Be careful when reading about VAX instructions, as they refer to the names of the VAX data types.

The VAX provides sixteen 32-bit registers. The VAX assembler uses the notation `r0`, `r1`, . . . , `r15` to refer to these registers, and we will stick to that notation. Alas, 4 of these 16 registers are effectively claimed by the instruction set architecture. For example, `r14` is the stack pointer (`sp`) and `r15` is the program counter (`pc`). Hence, `r15` cannot be used as a general-purpose register, and using `r14` is very difficult because it interferes with instructions that manipulate the stack. The other dedicated registers are `r12`, used as the argument pointer (`ap`), and `r13`, used as the frame pointer (`fp`); their purpose will become clear later. (Like MIPS, the VAX assembler accepts either the register number or the register name.)

VAX addressing modes include those discussed in Appendix A, which has all the MIPS addressing modes: *register, displacement, immediate*, and *PC-relative*. Moreover, all these modes can be used for jump addresses or for data addresses.

But that's not all the addressing modes. To reduce code size, the VAX has three lengths of addresses for displacement addressing: 8-bit, 16-bit, and 32-bit addresses called, respectively, *byte displacement, word displacement*, and *long displacement* addressing. Thus, an address can be not only as small as possible but also as large as necessary; large addresses need not be split, so there is no equivalent to the MIPS lui instruction (see Figure A.24 on page A-37).

Those are still not all the VAX addressing modes. Several have a *deferred* option, meaning that the object addressed is only the *address* of the real object, requiring another memory access to get the operand. This addressing mode is called *indirect addressing* in other machines. Thus, *register deferred, autoincrement deferred*, and *byte/word/long displacement deferred* are other addressing modes to choose from. For example, using the notation of the VAX assembler, r1 means the operand is register 1 and (`r1`) means the operand is the location in memory pointed to by `r1`.

There is yet another addressing mode. *Indexed addressing* automatically converts the value in an index operand to the proper byte address to add to the rest of the address. For a 32-bit word, we needed to multiply the index of a 4-byte quantity by 4 before adding it to a base address. Indexed addressing, called *scaled addressing* on some computers, automatically multiplies the index of a 4-byte quantity by 4 as part of the address calculation.

To cope with such a plethora of addressing options, the VAX architecture separates the specification of the addressing mode from the specification of the operation. Hence, the opcode supplies the operation and the number of operands, and each operand has its own addressing mode specifier. Figure K.50 shows the name, assembler notation, example, meaning, and length of the address specifier.

The VAX style of addressing means that an operation doesn't know where its operands come from; a VAX add instruction can have three operands in registers, three operands in memory, or any combination of registers and memory operands.

Addressing mode name	Syntax	Example	Meaning	Length of address specifier in bytes
Literal	#value	#–1	–1	1 (6-bit signed value)
Immediate	#value	#100	100	1 + length of the immediate
Register	rn	r3	r3	1
Register deferred	(rn)	(r3)	Memory[r3]	1
Byte/word/long displacement	Displacement (rn)	100(r3)	Memory[r3 + 100]	1 + length of the displacement
Byte/word/long displacement deferred	@displacement (rn)	@100(r3)	Memory[Memory [r3 + 100]]	1 + length of the displacement
Indexed (scaled)	Base mode [rx]	(r3)[r4]	Memory[r3 + r4 × d] (where d is data size in bytes)	1 + length of base addressing mode
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + d	1
Autodecrement	– (rn)	–(r3)	r3 = r3 – d; Memory[r3]	1
Autoincrement deferred	@(rn)+	@(r3)+	Memory[Memory[r3]]; r3 = r3 + d	1

Figure K.50 Definition and length of the VAX operand specifiers. The length of each addressing mode is 1 byte plus the length of any displacement or immediate field needed by the mode. Literal mode uses a special 2-bit tag and the remaining 6 bits encode the constant value. If the constant is too big, it must use the immediate addressing mode. Note that the length of an immediate operand is dictated by the length of the data type indicated in the opcode, not the value of the immediate. The symbol *d* in the last four modes represents the length of the data in bytes; *d* is 4 for 32-bit add.

Example How long is the following instruction?

```
addl3 r1,737(r2),(r3)[r4]
```

The name addl3 means a 32-bit add instruction with three operands. Assume the length of the VAX opcode is 1 byte.

Answer The first operand specifier—`r1`—indicates register addressing and is 1 byte long. The second operand specifier—`737(r2)`—indicates displacement addressing and has two parts: The first part is a byte that specifies the word displacement addressing mode and base register `(r2)`; the second part is the 2-byte-long displacement `(737)`. The third operand specifier—`(r3)[r4]`—also has two parts: The first byte specifies register deferred addressing mode `((r3))`, and the second byte specifies the Index register and the use of indexed addressing ([r4]). Thus, the total length of the instruction is 1 + (1) + (1 + 2) + (1 + 1) = 7 bytes.

In this example instruction, we show the VAX destination operand on the left and the source operands on the right, just as we show MIPS code. The VAX assembler actually expects operands in the opposite order, but we felt it would be less confusing to keep the destination on the left for both machines. Obviously, left or right orientation is arbitrary; the only requirement is consistency.

Elaboration Because the PC is 1 of the 16 registers that can be selected in a VAX addressing mode, 4 of the 22 VAX addressing modes are synthesized from other addressing modes. Using the PC as the chosen register in each case, immediate addressing is really autoincrement, PC-relative is displacement, absolute is autoincrement deferred, and relative deferred is displacement deferred.

Encoding VAX Instructions

Given the independence of the operations and addressing modes, the encoding of instructions is quite different from MIPS.

VAX instructions begin with a single byte opcode containing the operation and the number of operands. The operands follow the opcode. Each operand begins with a single byte, called the *address specifier*, that describes the addressing mode for that operand. For a simple addressing mode, such as register addressing, this byte specifies the register number as well as the mode (see the rightmost column in Figure K.50). In other cases, this initial byte can be followed by many more bytes to specify the rest of the address information.

As a specific example, let's show the encoding of the add instruction from the example on page K-24:

```
addl3 r1,737(r2),(r3)[r4]
```

Assume that this instruction starts at location 201.

Figure K.51 shows the encoding. Note that the operands are stored in memory in opposite order to the assembly code above. The execution of VAX instructions

Byte address	Contents at each byte	Machine code
201	Opcode containing addl3	$\mathrm{c1}_{\mathrm{hex}}$
202	Index mode specifier for [r4]	44_{hex}
203	Register indirect mode specifier for (r3)	63_{hex}
204	Word displacement mode specifier using r2 as base	$\mathrm{c2}_{\mathrm{hex}}$
205	The 16-bit constant 737	$\mathrm{e1}_{\mathrm{hex}}$
206		02_{hex}
207	Register mode specifier for r1	51_{hex}

Figure K.51 The encoding of the VAX instruction addl3 r1,737(r2),(r3)[r4], assuming it starts at address 201. To satisfy your curiosity, the right column shows the actual VAX encoding in hexadecimal notation. Note that the 16-bit constant 737_{ten} takes 2 bytes.

begins with fetching the source operands, so it makes sense for them to come first. Order is not important in fixed-length instructions like MIPS, since the source and destination operands are easily found within a 32-bit word.

The first byte, at location 201, is the opcode. The next byte, at location 202, is a specifier for the index mode using register `r4`. Like many of the other specifiers, the left 4 bits of the specifier give the mode and the right 4 bits give the register used in that mode. Since `addl3` is a 4-byte operation, `r4` will be multiplied by 4 and added to whatever address is specified next. In this case it is register deferred addressing using register `r3`. Thus, bytes 202 and 203 combined define the third operand in the assembly code.

The following byte, at address 204, is a specifier for word displacement addressing using register `r2` as the base register. This specifier tells the VAX that the following two bytes, locations 205 and 206, contain a 16-bit address to be added to `r2`.

The final byte of the instruction gives the destination operand, and this specifier selects register addressing using register `r1`.

Such variability in addressing means that a single VAX operation can have many different lengths; for example, an integer add varies from 3 bytes to 19 bytes. VAX implementations must decode the first operand before they can find the second, and so implementors are strongly tempted to take 1 clock cycle to decode each operand; thus, this sophisticated instruction set architecture can result in higher clock cycles per instruction, even when using simple addresses.

VAX Operations

In keeping with its philosophy, the VAX has a large number of operations as well as a large number of addressing modes. We review a few here to give the flavor of the machine.

Given the power of the addressing modes, the VAX *move* instruction performs several operations found in other machines. It transfers data between any two addressable locations and subsumes load, store, register-register moves, and memory-memory moves as special cases. The first letter of the VAX data type (b, w, l, f, q, d, g, c in Figure K.49) is appended to the acronym mov to determine the size of the data. One special move, called *move address*, moves the 32-bit *address* of the operand rather than the data. It uses the acronym `mova`.

The arithmetic operations of MIPS are also found in the VAX, with two major differences. First, the type of the data is attached to the name. Thus, `addb`, `addw`, and `addl` operate on 8-bit, 16-bit, and 32-bit data in memory or registers, respectively; MIPS has a single add instruction that operates only on the full 32-bit register. The second difference is that to reduce code size the add instruction specifies the number of unique operands; MIPS always specifies three even if one operand is redundant. For example, the MIPS instruction

```
add $1, $1, $2
```

takes 32 bits like all MIPS instructions, but the VAX instruction

```
addl2 r1, r2
```

uses `r1` for both the destination and a source, taking just 24 bits: 8 bits for the opcode and 8 bits each for the two register specifiers.

Number of Operations

Now we can show how VAX instruction names are formed:

$$(\text{operation})(\text{datatype})\left(\frac{2}{3}\right)$$

The operation add works with data types byte, word, long, float, and double and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

```
addb2   addw2   addl2   addf2   addd2
addb3   addw3   addl3   addf3   addd3
```

Accounting for all addressing modes (but ignoring register numbers and immediate values) and limiting to just byte, word, and long, there are more than 30,000 versions of integer add in the VAX; MIPS has just 4!

Another reason for the large number of VAX instructions is the instructions that either replace sequences of instructions or take fewer bytes to represent a single instruction. Here are four such examples (* means the data type):

VAX operation	Example	Meaning
`clr*`	`clrl r3`	`r3 = 0`
`inc*`	`incl r3`	`r3 = r3+1`
`dec*`	`decl r3`	`r3 = r3-1`
`push*`	`pushl r3`	`sp = sp-4;Memory[sp]=r3;`

The *push* instruction in the last row is exactly the same as using the move instruction with autodecrement addressing on the stack pointer:

```
movl -(sp), r3
```

Brevity is the advantage of pushl: It is 1 byte shorter since sp is implied.

Branches, Jumps, and Procedure Calls

The VAX branch instructions are related to the arithmetic instructions because the branch instructions rely on *condition codes*. Condition codes are set as a side effect of an operation, and they indicate whether the result is positive, negative, or zero or if an overflow occurred. Most instructions set the VAX condition codes according to their result; instructions without results, such as branches, do not. The VAX condition codes are N (Negative), Z (Zero), V (oVerflow), and C (Carry). There is also a *compare* instruction `cmp*` just to set the condition codes for a subsequent branch.

The VAX branch instructions include all conditions. Popular branch instructions include `beql(=)`, `bneq(≠)`, `blss(<)`, `bleq(≤)`, `bgtr(>)`, and `bgeq(≥)`, which do just what you would expect. There are also unconditional branches whose name is determined by the size of the PC-relative offset. Thus, `brb` (*branch byte*) has an 8-bit displacement, and `brw` (*branch word*) has a 16-bit displacement.

The final major category we cover here is the procedure *call and return* instructions. Unlike the MIPS architecture, these elaborate instructions can take dozens of clock cycles to execute. The next two sections show how they work, but we need to explain the purpose of the pointers associated with the stack manipulated by `calls` and `ret`. The *stack pointer*, `sp`, is just like the stack pointer in MIPS; it points to the top of the stack. The *argument pointer*, `ap`, points to the base of the list of arguments or parameters in memory that are passed to the procedure. The *frame pointer*, `fp`, points to the base of the local variables of the procedure that are kept in memory (the *stack frame*). The VAX call and return instructions manipulate these pointers to maintain the stack in proper condition across procedure calls and to provide convenient base registers to use when accessing memory operands. As we shall see, call and return also save and restore the general-purpose registers as well as the program counter. Figure K.52 gives a further sampling of the VAX instruction set.

An Example to Put It All Together: swap

To see programming in VAX assembly language, we translate two C procedures, `swap` and `sort`. The C code for `swap` is reproduced in Figure K.53. The next section covers `sort`.

We describe the `swap` procedure in three general steps of assembly language programming:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

Instruction type	Example	Instruction meaning
Data transfers	Move data between byte, half-word, word, or double-word operands; * is data type	
	`mov*`	Move between two operands
	`movzb*`	Move a byte to a half word or word, extending it with zeros
	`mova*`	Move the 32-bit address of an operand; data type is last
	`push*`	Push operand onto stack
Arithmetic/logical	Operations on integer or logical bytes, half words (16 bits), words (32 bits); * is data type	
	`add*_`	Add with 2 or 3 operands
	`cmp*`	Compare and set condition codes
	`tst*`	Compare to zero and set condition codes
	`ash*`	Arithmetic shift
	`clr*`	Clear
	`cvtb*`	Sign-extend byte to size of data type
Control	Conditional and unconditional branches	
	`beql, bneq`	Branch equal, branch not equal
	`bleq, bgeq`	Branch less than or equal, branch greater than or equal
	`brb, brw`	Unconditional branch with an 8-bit or 16-bit address
	`jmp`	Jump using any addressing mode to specify target
	`aobleq`	Add one to operand; branch if result ≤ second operand
	`case_`	Jump based on case selector
Procedure	Call/return from procedure	
	`calls`	Call procedure with arguments on stack (see "A Longer Example: `sort`" on page K-33)
	`callg`	Call procedure with FORTRAN-style parameter list
	`jsb`	Jump to subroutine, saving return address (like MIPS jal)
	`ret`	Return from procedure call
Floating point	Floating-point operations on D, F, G, and H formats	
	`addd_`	Add double-precision D-format floating numbers
	`subd_`	Subtract double-precision D-format floating numbers
	`mulf_`	Multiply single-precision F-format floating point
	`polyf`	Evaluate a polynomial using table of coefficients in F format
Other	Special operations	
	`crc`	Calculate cyclic redundancy check
	`insque`	Insert a queue entry into a queue

Figure K.52 Classes of VAX instructions with examples. The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in `addd_`, means there are 2-operand (`addd2`) and 3-operand (`addd3`) forms of this instruction.

```
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k + 1];
   v[k + 1] = temp;
}
```

Figure K.53 A C procedure that swaps two locations in memory. This procedure will be used in the sorting example in the next section.

The VAX code for these procedures is based on code produced by the VMS C compiler using optimization.

Register Allocation for `swap`

In contrast to MIPS, VAX parameters are normally allocated to memory, so this step of assembly language programming is more properly called "variable allocation." The standard VAX convention on parameter passing is to use the stack. The two parameters, `v[]` and `k`, can be accessed using register ap, the argument pointer: The address `4(ap)` corresponds to `v[]` and `8(ap)` corresponds to k. Remember that with byte addressing the address of sequential 4-byte words differs by 4. The only other variable is `temp`, which we associate with register `r3`.

Code for the Body of the Procedure `swap`

The remaining lines of C code in `swap` are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

Since this program uses `v[]` and `k` several times, to make the programs run faster the VAX compiler first moves both parameters into registers:

```
movl r2, 4(ap) ;r2 = v[]
movl r1, 8(ap) ;r1 = k
```

Note that we follow the VAX convention of using a semicolon to start a comment; the MIPS comment symbol # represents a constant operand in VAX assembly language.

The VAX has indexed addressing, so we can use index k without converting it to a byte address. The VAX code is then straightforward:

```
movl  r3, (r2)[r1]       ;r3 (temp) = v[k]
addl3 r0, #1,8(ap)       ;r0 = k + 1
movl  (r2)[r1],(r2)[r0] ;v[k] = v[r0] (v[k + 1])
movl  (r2)[r0],r3       ;v[k + 1] = r3 (temp)
```

Unlike the MIPS code, which is basically two loads and two stores, the key VAX code is one memory-to-register move, one memory-to-memory move, and one register-to-memory move. Note that the `addl3` instruction shows the flexibility of the VAX addressing modes: It adds the constant 1 to a memory operand and places the result in a register.

Now we have allocated storage and written the code to perform the operations of the procedure. The only missing item is the code that preserves registers across the routine that calls `swap`.

Preserving Registers across Procedure Invocation of `swap`

The VAX has a pair of instructions that preserve registers, `calls` and `ret`. This example shows how they work.

The VAX C compiler uses a form of callee convention. Examining the code above, we see that the values in registers `r0`, `r1`, `r2`, and `r3` must be saved so that they can later be restored. The `calls` instruction expects a 16-bit mask at the beginning of the procedure to determine which registers are saved: if bit *i* is set in the mask, then register *i* is saved on the stack by the `calls` instruction. In addition, `calls` saves this mask on the stack to allow the return instruction (`ret`) to restore the proper registers. Thus, the calls executed by the caller does the saving, but the callee sets the call mask to indicate what should be saved.

One of the operands for `calls` gives the number of parameters being passed, so that `calls` can adjust the pointers associated with the stack: the argument pointer (`ap`), frame pointer (`fp`), and stack pointer (`sp`). Of course, `calls` also saves the program counter so that the procedure can return!

Thus, to preserve these four registers for swap, we just add the mask at the beginning of the procedure, letting the calls instruction in the caller do all the work:

```
.word ^m<r0,r1,r2,r3> ;set bits in mask for 0,1,2,3
```

This directive tells the assembler to place a 16-bit constant with the proper bits set to save registers `r0` through `r3`.

The return instruction undoes the work of `calls`. When finished, `ret` sets the stack pointer from the current frame pointer to pop everything `calls` placed on the stack. Along the way, it restores the register values saved by `calls`, including those marked by the mask and old values of the `fp`, `ap`, and `pc`.

To complete the procedure `swap`, we just add one instruction:

```
ret   ;restore registers and return
```

The Full Procedure `swap`

We are now ready for the whole routine. Figure K.54 identifies each block of code with its purpose in the procedure, with the MIPS code on the left and the VAX code on the right. This example shows the advantage of the scaled indexed addressing and the sophisticated call and return instructions of the VAX in reducing the number of lines of code. The 17 lines of MIPS assembly code became 8 lines of VAX assembly code. It also shows that passing parameters in memory results in extra memory accesses.

Keep in mind that the number of instructions executed is not the same as performance; the fallacy on page K-38 makes this point.

Note that VAX software follows a convention of treating registers `r0` and `r1` as temporaries that are not saved across a procedure call, so the VMS C compiler does include registers `r0` and `r1` in the register saving mask. Also, the C compiler should have used `r1` instead of `8(ap)` in the `addl3` instruction; such examples inspire computer architects to try to write compilers!

MIPS versus VAX

Saving register

```
swap: addi   $29,$29, -12      swap: .word ^m<r0,r1,r2,r3>
      sw      $2, 0($29)
      sw     $15, 4($29)
      sw     $16, 8($29)
```

Procedure body

```
      muli   $2, $5,4                 movl   r2, 4(a)
      add    $2, $4,$2                movl   r1, 8(a)
      lw     $15, 0($2)               movl   r3, (r2)[r1]
      lw     $16, 4($2)               addl3  r0, #1,8(ap)
      sw     $16, 0($2)               movl   (r2)[r1],(r2)[r0]
      sw     $15, 4($2)               movl   (r2)[r0],r3
```

Restoring registers

```
      lw      $2, 0($29)
      lw     $15, 4($29)
      lw     $16, 8($29)
      addi   $29,$29, 12
```

Procedure return

```
      jr     $31                      ret
```

Figure K.54 MIPS versus VAX assembly code of the procedure swap in Figure K.53 on page K-30.

A Longer Example: sort

We show the longer example of the sort procedure. Figure K.55 shows the C version of the program. Once again we present this procedure in several steps, concluding with a side-by-side comparison to MIPS code.

Register Allocation for `sort`

The two parameters of the procedure sort, v and n, are found in the stack in locations 4(ap) and 8(ap), respectively. The two local variables are assigned to registers: i to r6 and j to r4. Because the two parameters are referenced frequently in the code, the VMS C compiler copies the *address* of these parameters into registers upon entering the procedure:

```
moval  r7,8(ap)  ;move  address  of  n  into  r7
moval  r5,4(ap)  ;move  address  of  v  into  r5
```

It would seem that moving the *value* of the operand to a register would be more useful than its address, but once again we bow to the decision of the VMS C compiler. Apparently the compiler cannot be sure that v and n don't overlap in memory.

Code for the Body of the sort Procedure

The procedure body consists of two nested *for* loops and a call to swap, which includes parameters. Let's unwrap the code from the outside to the middle.

The Outer Loop

The first translation step is the first for loop:

```
for (i = 0; i < n; i = i + 1) {
```

Recall that the C for statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to `0`, the first part of the for statement:

```
clrl   r6    ;i = 0
```

```
sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i = i + 1) {
  for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1)
    { swap(v,j);
   }
 }
}
```

Figure K.55 A C procedure that performs a bubble sort on the array v.

It also takes just one instruction to increment `i`, the last part of the for:

```
incl    r6    ;i = i + 1
```

The loop should be exited if `i` < `n` is *false,* or said another way, exit the loop if `i` ≥ `n`. This test takes two instructions:

```
for1tst: cmpl r6,(r7) ;compare r6 and memory[r7] (i:n)
         bgeq exit1   ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
```

Note that `cmpl` sets the condition codes for use by the conditional branch instruction `bgeq`.

The bottom of the loop just jumps back to the loop test:

```
         brb  for1tst  ;branch to test of outer loop
exit1:
```

The skeleton code of the first for loop is then

```
         clrl  r6       ;i = 0
for1tst: cmpl  r6,(r7)  ;compare r6 and memory[r7] (i:n)
         bgeq exit1     ;go to exit1 if r6 ≥ mem[r7] (i ≥ n)
               ...
               (body of first for loop)
               ...
         incl  r6       ;i = i + 1
         brb   for1tst  ;branch to test of outer loop
exit1:
```

The Inner Loop

The second for loop is

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) {
```

The initialization portion of this loop is again one instruction:

```
subl3    r4,r6,#1    ;j = i - 1
```

The decrement of j is also one instruction:

```
decl     r4          ;j = j - 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails `(j < 0)`:

```
for2tst:blss     exit2      ;go to exit2 if r4 < 0 (j < 0)
```

Notice that there is no explicit comparison. The lack of comparison is a benefit of condition codes, with the conditions being set as a side effect of the prior instruction. This branch skips over the second condition test.

The second test exits if `v[j] > v[j + 1]` is false, or exits if `v[j] ≤ v[j + 1]`. First we load `v` and put `j + 1` into registers:

```
movl    r3,(r5)    ;r3 = Memory[r5] (r3 = v)
addl3   r2,r4,#1   ;r2 = r4 + 1 (r2 = j + 1)
```

Register indirect addressing is used to get the operand pointed to by r5.

Once again the index addressing mode means we can use indices without converting to the byte address, so the two instructions for `v[j] ≤ v[j + 1]` are

```
cmpl (r3)[r4],(r3)[r2]    ;v[r4] : v[r2] (v[j]:v[j + 1])
bleq exit2                ;go to exit2 if v[j] ≤ v[j + 1]
```

The bottom of the loop jumps back to the full loop test:

```
brb      for2tst #      jump to test of inner loop
```

Combining the pieces, the second for loop looks like this:

```
          subl3 r4,r6, #1     ;j = i - 1
for2tst:  blss  exit2         ;go to exit2 if r4 < 0 (j < 0)
          movl  r3,(r5)       ;r3 = Memory[r5] (r3 = v)
          addl3 r2,r4,#1      ;r2 = r4 + 1 (r2 = j + 1)
          cmpl  (r3)[r4],(r3)[r2];v[r4] : v[r2]
          bleq  exit2         ;go to exit2 if v[j] ð [j+1]
                ...
                (body of second for loop) ...
          decl  r4            ;j = j - 1
          brb   for2tst       ;jump to test of inner loop
exit2:
```

Notice that the instruction `blss` (at the top of the loop) is testing the condition codes based on the new value of `r4 (j)`, set either by the `subl3` before entering the loop or by the `decl` at the bottom of the loop.

The Procedure Call

The next step is the body of the second for loop:

```
swap(v,j);
```

Calling swap is easy enough:

```
calls     #2,swap
```

The constant 2 indicates the number of parameters pushed on the stack.

Passing Parameters

The C compiler passes variables on the stack, so we pass the parameters to `swap` with these two instructions:

```
pushl    (r5)          ;first swap parameter is v
pushl     r4           ;second swap parameter is j
```

Register indirect addressing is used to get the operand of the first instruction.

Preserving Registers across Procedure Invocation of sort

The only remaining code is the saving and restoring of registers using the callee save convention. This procedure uses registers `r2` through `r7`, so we add a mask with those bits set:

```
    .word ^m<r2,r3,r4,r5,r6,r7>; set mask for registers 2-7
```

Since ret will undo all the operations, we just tack it on the end of the procedure.

The Full Procedure sort

Now we put all the pieces together in Figure K.56. To make the code easier to follow, once again we identify each block of code with its purpose in the procedure and list the MIPS and VAX code side by side. In this example, 11 lines of the sort procedure in C become the 44 lines in the MIPS assembly language and 20 lines in VAX assembly language. The biggest VAX advantages are in register saving and restoring and indexed addressing.

Fallacies and Pitfalls

> *The ability to simplify means to eliminate the unnecessary so that the necessary may speak.*
>
> **Hans Hoffman**
> *Search for the Real* (1967)

Fallacy *It is possible to design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at one time later look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code size efficiency and underestimated how important ease of decoding and pipelining would be 10 years later. And, almost all architectures eventually succumb to the lack of sufficient address space. Avoiding these problems in the long run, however, would probably mean compromising the efficiency of the architecture in the short run.

Fallacy *An architecture with flaws cannot be successful.*

The IBM 360 is often criticized in the literature—the branches are not PC-relative, and the address is too small in displacement addressing. Yet, the machine has been an enormous success because it correctly handled several new problems. First, the architecture has a large amount of address space. Second, it is byte addressed and handles bytes well. Third, it is a general-purpose register machine. Finally, it is simple enough to be efficiently implemented across a wide performance and cost range.

The Intel 8086 provides an even more dramatic example. The 8086 architecture is the only widespread architecture in existence today that is not truly a general-purpose register machine. Furthermore, the segmented address space of the 8086 causes major problems for both programmers and compiler writers. Nevertheless, the 8086 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

MIPS versus VAX

```
                              Saving registers
             sort:    addi $29,$29, -36      sort:  .word ^m<r2,r3,r4,r5,r6,r7>
                      sw   $15, 0($29)
                      sw   $16, 4($29)
                      sw   $17, 8($29)
                      sw   $18,12($29)
                      sw   $19,16($29)
                      sw   $20,20($29)
                      sw   $24,24($29)
                      sw   $25,28($29)
                      sw   $31,32($29)
                              Procedure body
Move parameters       move $18, $4                    moval  r7,8(ap)
                      move $20, $5                    moval  r5,4(ap)
Outer loop            add  $19, $0, $0                clrl   r6
             for1tst: slt  $8, $19, $20      for1tst: cmpl   r6,(r7)
                      beq  $8, $0, exit1              bgeq   exit1
Inner loop            addi $17, $19, -1               subl3  r4,r6,#1
             for2tst: slti $8, $17, 0        for2tst:
                      bne  $8, $0, exit2              blss   exit2
                      muli $15, $17, 4                movl   r3,(r5)
                      add  $16, $18, $15
                      lw   $24, 0($16)
                      lw   $25, 4($16)                addl3  r2,r4,#1
                      slt  $8, $25, $24               cmpl   (r3)[r4],(r3)[r2]
                      beq  $8, $0, exit2              bleq   exit2
Pass parameters       move $4, $18                    pushl  (r5)
and call              move $5, $17                    pushl  r4
                      jal  swap                       calls  #2,swap
Inner loop            addi $17, $17, -1               decl   r4
                      j    for2tst                    brb    for2tst
Outer loop   exit2:   addi $19, $19, 1       exit2:   incl   r6
                      j    for1tst                    brb    for1tst
                              Restoring registers
             exit1:   lw   $15,0($29)
                      lw   $16, 4($29)
                      lw   $17, 8($29)
                      lw   $18,12($29)
                      lw   $19,16($29)
                      lw   $20,20($29)
                      lw   $24,24($29)
                      lw   $25,28($29)
                      lw   $31,32($29)
                      addi $29,$29, 36
                              Procedure return
                      jr   $31               exit1:   ret
```

Figure K.56 MIPS32 versus VAX assembly version of procedure sort **in Figure K.55 on page K-33.**

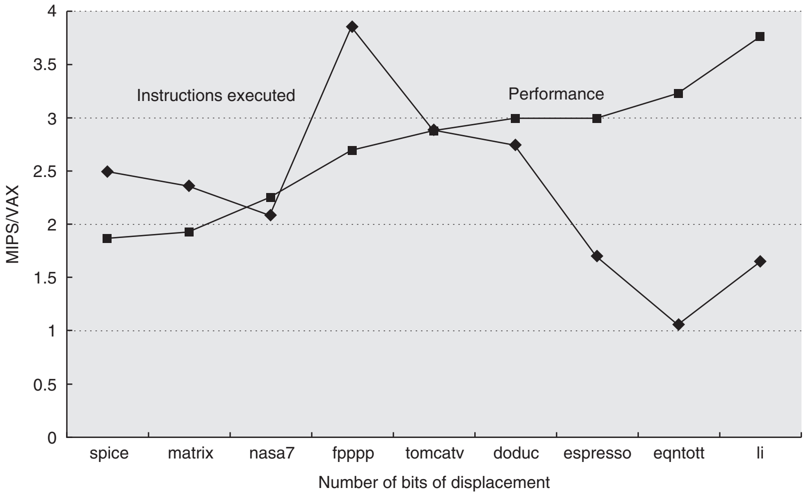


Figure K.57 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from "Performance from Architecture: Comparing a RISC and CISC with Similar Hardware Organization," by D. Bhandarkar and D. Clark, in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems IV*, 1991.)

Fallacy *The architecture that executes fewer instructions is faster.*

Designers of VAX machines performed a quantitative comparison of VAX and MIPS for implementations with comparable organizations, the VAX 8700 and the MIPS M2000. Figure K.57 shows the ratio of the number of instructions executed and the ratio of performance measured in clock cycles. MIPS executes about twice as many instructions as the VAX while the MIPS M2000 has almost three times the performance of the VAX 8700.

Concluding Remarks

The Virtual Address eXtension of the PDP-11 architecture … provides a virtual address of about 4.3 gigabytes which, even given the rapid improvement of memory technology, should be adequate far into the future.

William Strecker

"VAX-11/780—A Virtual Address Extension to the PDP-11 Family,"
AFIPS Proc., *National Computer Conference* (1978)

We have seen that instruction sets can vary quite dramatically, both in how they access operands and in the operations that can be performed by a single instruction. Figure K.58 compares instruction usage for both architectures for two programs; even very different architectures behave similarly in their use of instruction classes.

Program	Machine	Branch	Arithmetic/ logical	Data transfer	Floating point	Totals
gcc	VAX	30%	40%	19%		89%
	MIPS	24%	35%	27%		86%
spice	VAX	18%	23%	15%	23%	79%
	MIPS	4%	29%	35%	15%	83%

Figure K.58 The frequency of instruction distribution for two programs on VAX and MIPS.

A product of its time, the VAX emphasis on code density and complex operations and addressing modes conflicts with the current emphasis on easy decoding, simple operations and addressing modes, and pipelined performance.

With more than 600,000 sold, the VAX architecture has had a very successful run. In 1991, DEC made the transition from VAX to Alpha.

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes, which are independent of the data types and even the number of unique operands. Thus, a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes.

Exercises

K.1 [3] <K.4> The following VAX instruction decrements the location pointed to be register r5:

```
decl (r5)
```

What is the single MIPS instruction, or if it cannot be represented in a single instruction, the shortest sequence of MIPS instructions, that performs the same operation? What are the lengths of the instructions on each machine?

K.2 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction clears a location using autoincrement deferred addressing:

```
clrl @(r5)+
```

K.3 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction adds 1 to register r5, placing the sum back in register r5, compares the sum to register r6, and then branches to L1 if r5 < r6:

```
aoblss r6, r5, L1 # r5 = r5 + 1; if (r5 < r6) goto L1.
```

K.4 [5] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
a = b + 100;
```

Assume a corresponds to register r3 and b corresponds to register r4.

K.5 [10] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

```
x[i + 1] = x[i] + c;
```

Assume c corresponds to register r3, i to register r4, and x is an array of 32-bit words beginning at memory location $4{,}000{,}000_{\text{ten}}$.

K.5 The IBM 360/370 Architecture for Mainframe Computers

Introduction

The term “computer architecture” was coined by IBM in 1964 for use with the IBM 360. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the instruction set. They believed that a family of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at the time. IBM, even though it was the leading company in the industry, had five different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of *architecture* was

> … the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term “machine language programmer” meant that compatibility would hold, even in assembly language, while “timing independent” allowed different implementations.

The IBM 360 was introduced in 1964 with six models and a 25:1 performance ratio. Amdahl, Blaauw, and Brooks [1964] discussed the architecture of the IBM 360 and the concept of permitting multiple object-code-compatible implementations. The notion of an instruction set architecture as we understand it today was the most important aspect of the 360. The architecture also introduced several important innovations, now in wide use:

1. 32-bit architecture
2. Byte-addressable memory with 8-bit bytes
3. 8-, 16-, 32-, and 64-bit data sizes
4. 32-bit single-precision and 64-bit double-precision floating-point data

In 1971, IBM shipped the first System/370 (models 155 and 165), which included a number of significant extensions of the 360, as discussed by Case and Padegs [1978], who also discussed the early history of System/360. The most important addition was virtual memory, though virtual memory 370 s did not ship until 1972, when a virtual memory operating system was ready. By 1978, the high-end 370 was several hundred times faster than the low-end 360 s shipped 10 years earlier. In 1984, the 24-bit addressing model built into the IBM 360 needed to be abandoned, and the 370-XA (eXtended Architecture) was introduced. While old 24-bit programs could be supported without change, several instructions could not function in the same manner when extended to a 32-bit addressing model (31-bit addresses supported) because they would not produce 31-bit addresses. Converting the operating system, which was written mostly in assembly language, was no doubt the biggest task.

Several studies of the IBM 360 and instruction measurement have been made. Shustek’s thesis [1978] is the best known and most complete study of the 360/370 architecture. He made several observations about instruction set complexity that were not fully appreciated until some years later. Another important study of the 360 is the Toronto study by Alexander and Wortman [1975] done on an IBM 360 using 19 XPL programs.

System/360 Instruction Set

The 360 instruction set is shown in the following tables, organized by instruction type and format. System/370 contains 15 additional user instructions.

Integer/Logical and Floating-Point R-R Instructions

The * indicates the instruction is floating point, and may be either D (double precision) or E (single precision).

Instruction	Description
`ALR`	Add logical register
`AR`	Add register
`A*R`	FP addition
`CLR`	Compare logical register
`CR`	Compare register
`C*R`	FP compare
`DR`	Divide register
`D*R`	FP divide
`H*R`	FP halve
`LCR`	Load complement register
`LC*R`	Load complement
`LNR`	Load negative register
`LN*R`	Load negative
`LPR`	Load positive register
`LP*R`	Load positive
`LR`	Load register
`L*R`	Load FP register
`LTR`	Load and test register
`LT*R`	Load and test FP register
`MR`	Multiply register
`M*R`	FP multiply
`NR`	And register
`OR`	Or register
`SLR`	Subtract logical register
`SR`	Subtract register
`S*R`	FP subtraction
`XR`	Exclusive or register

Branches and Status Setting R-R Instructions

These are R-R format instructions that either branch or set some system status; several of them are privileged and legal only in supervisor mode.

Instruction	**Description**
`BALR`	Branch and link
`BCTR`	Branch on count
`BCR`	Branch/condition
`ISK`	Insert key
`SPM`	Set program mask
`SSK`	Set storage key
`SVC`	Supervisor call

Branches/Logical and Floating-Point Instructions—RX Format

These are all RX format instructions. The symbol "+" means either a word operation (and then stands for nothing) or H (meaning half word); for example, `A+` stands for the two opcodes `A` and `AH`. The "*" represents `D` or `E`, standing for double- or single-precision floating point.

Instruction	**Description**
`A+`	Add
`A*`	FP add
`AL`	Add logical
`C+`	Compare
`C*`	FP compare
`CL`	Compare logical
`D`	Divide
`D*`	FP divide
`L+`	Load
`L*`	Load FP register
`M+`	Multiply
`M*`	FP multiply
`N`	And
`O`	Or
`S+`	Subtract
`S*`	FP subtract
`SL`	Subtract logical
`ST+`	Store
`ST*`	Store FP register
`X`	Exclusive or

Branches and Special Loads and Stores—RX Format

Instruction	**Description**
`BAL`	Branch and link
`BC`	Branch condition
`BCT`	Branch on count
`CVB`	Convert-binary
`CVD`	Convert-decimal
`EX`	Execute
`IC`	Insert character
`LA`	Load address
`STC`	Store character

RS and SI Format Instructions

These are the RS and SI format instructions. The symbol "`*`" may be `A` (arithmetic) or `L` (logical).

Instruction	**Description**
`BXH`	Branch/high
`BXLE`	Branch/low-equal
`CLI`	Compare logical immediate
`HIO`	Halt I/O
`LPSW`	Load PSW
`LM`	Load multiple
`MVI`	Move immediate
`NI`	And immediate
`OI`	Or immediate
`RDD`	Read direct
`SIO`	Start I/O
`SL*`	Shift left A/L
`SLD*`	Shift left double A/L
`SR*`	Shift right A/L
`SRD*`	Shift right double A/L
`SSM`	Set system mask
`STM`	Store multiple
`TCH`	Test channel
`TIO`	Test I/O
`TM`	Test under mask
`TS`	Test-and-set
`WRD`	Write direct
`XI`	Exclusive or immediate

SS Format Instructions

These are add decimal or string instructions.

Instruction	Description
`AP`	Add packed
`CLC`	Compare logical chars
`CP`	Compare packed
`DP`	Divide packed
`ED`	Edit
`EDMK`	Edit and mark
`MP`	Multiply packed
`MVC`	Move character
`MVN`	Move numeric
`MVO`	Move with offset
`MVZ`	Move zone
`NC`	And characters
`OC`	Or characters
`PACK`	Pack (Character → decimal)
`SP`	Subtract packed
`TR`	Translate
`TRT`	Translate and test
`UNPK`	Unpack
`XC`	Exclusive or characters
`ZAP`	Zero and add packed

360 Detailed Measurements

Figure K.59 shows the frequency of instruction usage for four IBM 360 programs.

Instruction	PLIC	FORTGO	PLIGO	COBOLGO	Average
Control	**32%**	**13%**	**5%**	**16%**	**16%**
BC, BCR	28%	13%	5%	14%	15%
BAL, BALR	3%			2%	1%
Arithmetic/logical	**29%**	**35%**	**29%**	**9%**	**26%**
A, AR	3%	17%	21%		10%
SR	3%	7%			3%
SLL		6%	3%		2%
LA	8%	1%	1%		2%
CLI	7%				2%
NI				7%	2%
C	5%	4%	4%	0%	3%
TM	3%	1%		3%	2%
MH			2%		1%
Data transfer	**17%**	**40%**	**56%**	**20%**	**33%**
L, LR	7%	23%	28%	19%	19%
MVI	2%		16%	1%	5%
ST	3%		7%		3%
LD		7%	2%		2%
STD		7%	2%		2%
LPDR		3%			1%
LH	3%				1%
IC	2%				1%
LTR		1%			0%
Floating point		**7%**			**2%**
AD		3%			1%
MDR		3%			1%
Decimal, string	**4%**			**40%**	**11%**
MVC	4%			7%	3%
AP				11%	3%
ZAP				9%	2%
CVD				5%	1%
MP				3%	1%
CLC				3%	1%
CP				2%	1%
ED				1%	0%
Total	**82%**	**95%**	**90%**	**85%**	**88%**

Figure K.59 Distribution of instruction execution frequencies for the four 360 programs. All instructions with a frequency of execution greater than 1.5% are included. Immediate instructions, which operate on only a single byte, are included in the section that characterized their operation, rather than with the long character-string versions of the same operation. By comparison, the average frequencies for the major instruction classes of the VAX are 23% (control), 28% (arithmetic), 29% (data transfer), 7% (floating point), and 9% (decimal). Once again, a 1% entry in the average column can occur because of entries in the constituent columns. These programs are a compiler for the programming language PL-I and runtime systems for the programming languages FORTRAN, PL/I, and Cobol.

K.6 Historical Perspective and References

Section L.4 (available online) features a discussion on the evolution of instruction sets and includes references for further reading and exploration of related topics.

Acknowledgments

We would like to thank the following people for comments on drafts of this survey: Professor Steven B. Furber, University of Manchester; Dr. Dileep Bhandarkar, Intel Corporation; Dr. Earl Killian, Silicon Graphics/MIPS; and Dr. Hiokazu Takata, Mitsubishi Electric Corporation.

L

Advanced Concepts on Address Translation

by Abhishek Bhattacharjee

Appendix L is available online at https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1

M.1 Introduction M-2
M.2 The Early Development of Computers (Chapter 1) M-2
M.3 The Development of Memory Hierarchy and Protection (Chapter 2 and Appendix B) M-9
M.4 The Evolution of Instruction Sets (Appendices A, J, and K) M-17
M.5 The Development of Pipelining and Instruction-Level Parallelism (Chapter 3 and Appendices C and H) M-27
M.6 The Development of SIMD Supercomputers, Vector Computers, Multimedia SIMD Instruction Extensions, and Graphical Processor Units (Chapter 4) M-45
M.7 The History of Multiprocessors and Parallel Processing (Chapter 5 and Appendices F, G, and I) M-55
M.8 The Development of Clusters (Chapter 6) M-74
M.9 Historical Perspectives and References M-79
M.10 The History of Magnetic Storage, RAID, and I/O Buses (Appendix D) M-84

M

Historical Perspectives and References

If … history … teaches us anything, it is that man in his quest for knowledge and progress is determined and cannot be deterred.

John F. Kennedy
Address at Rice University (1962)

Those who cannot remember the past are condemned to repeat it.

George Santayana
The Life of Reason (1905), Vol. 2, Chapter 3

M.1 Introduction

This appendix provides historical background on some of the key ideas presented in the chapters. We may trace the development of an idea through a series of machines or describe significant projects. If you are interested in examining the initial development of an idea or machine or are interested in further reading, references are provided at the end of each section.

Section M.2 starts us off with the invention of the digital computer and corresponds to Chapter 1. Section M.3, on memory hierarchy, corresponds to Chapter 2 and Appendix B. Section M.4, on instruction set architecture, covers Appendices A, J, and K. Section M.5, on pipelining and instruction-level parallelism, corresponds to Chapter 3 and Appendices C and H. Section M.6, on data-level parallelism in vector, SIMD, and GPU architectures, corresponds to Chapter 4. Section M.7, on multiprocessors and parallel programming, covers Chapter 5 and Appendices F, G, and I. Section M.8, on the development of clusters, covers Chapter 6. Finally, Section M.9, on I/O, corresponds to Appendix D.

M.2 The Early Development of Computers (Chapter 1)

In this historical section, we discuss the early development of digital computers and the development of performance measurement methodologies.

The First General-Purpose Electronic Computers

J. Presper Eckert and John Mauchly at the Moore School of the University of Pennsylvania built the world's first fully operational electronic general-purpose computer. This machine, called ENIAC (Electronic Numerical Integrator and Calculator), was funded by the U.S. Army and became operational during World War II, but it was not publicly disclosed until 1946. ENIAC was used for computing artillery firing tables. The machine was enormous—100 feet long, 8½ feet high, and several feet wide. Each of the 20 ten-digit registers was 2 feet long. In total, there were 18,000 vacuum tubes.

Although the size was three orders of magnitude bigger than the size of the average machines built today, it was more than five orders of magnitude slower, with an add taking 200 microseconds. The ENIAC provided conditional jumps and was programmable, which clearly distinguished it from earlier calculators. Programming was done manually by plugging up cables and setting switches and required from a half hour to a whole day. Data were provided on punched cards. The ENIAC was limited primarily by a small amount of storage and tedious programming.

In 1944, John von Neumann was attracted to the ENIAC project. The group wanted to improve the way programs were entered and discussed storing programs as numbers; von Neumann helped crystallize the ideas and wrote a memo proposing a stored-program computer called EDVAC (Electronic Discrete Variable

Automatic Computer). Herman Goldstine distributed the memo and put von Neumann's name on it, much to the dismay of Eckert and Mauchly, whose names were omitted. This memo has served as the basis for the commonly used term *von Neumann computer*. Several early inventors in the computer field believe that this term gives too much credit to von Neumann, who conceptualized and wrote up the ideas, and too little to the engineers, Eckert and Mauchly, who worked on the machines. Like most historians, your authors (winners of the 2000 IEEE von Neumann Medal) believe that all three individuals played a key role in developing the stored-program computer. Von Neumann's role in writing up the ideas, in generalizing them, and in thinking about the programming aspects was critical in transferring the ideas to a wider audience.

In 1946, Maurice Wilkes of Cambridge University visited the Moore School to attend the latter part of a series of lectures on developments in electronic computers. When he returned to Cambridge, Wilkes decided to embark on a project to build a stored-program computer named EDSAC (Electronic Delay Storage Automatic Calculator). (The EDSAC used mercury delay lines for its memory; hence, the phrase "delay storage" in its name.) The EDSAC became operational in 1949 and was the world's first full-scale, operational, stored-program computer [Wilkes, Wheeler, and Gill 1951; Wilkes 1985, 1995]. (A small prototype called the Mark I, which was built at the University of Manchester and ran in 1948, might be called the first operational stored-program machine.) The EDSAC was an accumulator-based architecture. This style of instruction set architecture remained popular until the early 1970s. (Appendix A starts with a brief summary of the EDSAC instruction set.)

In 1947, Mauchly took the time to help found the Association for Computing Machinery. He served as the ACM's first vice-president and second president. That same year, Eckert and Mauchly applied for a patent on electronic computers. The dean of the Moore School, by demanding that the patent be turned over to the university, may have helped Eckert and Mauchly conclude that they should leave. Their departure crippled the EDVAC project, which did not become operational until 1952.

Goldstine left to join von Neumann at the Institute for Advanced Study at Princeton in 1946. Together with Arthur Burks, they issued a report based on the 1944 memo [Burks, Goldstine, and von Neumann 1946]. The paper led to the IAS machine built by Julian Bigelow at Princeton's Institute for Advanced Study. It had a total of 1024 40-bit words and was roughly 10 times faster than ENIAC. The group thought about uses for the machine, published a set of reports, and encouraged visitors. These reports and visitors inspired the development of a number of new computers, including the first IBM computer, the 701, which was based on the IAS machine. The paper by Burks, Goldstine, and von Neumann was incredible for the period. Reading it today, you would never guess this landmark paper was written more than 50 years ago, as most of the architectural concepts seen in modern computers are discussed there (e.g., see the quote at the beginning of Chapter 2).

In the same time period as ENIAC, Howard Aiken was designing an electromechanical computer called the Mark-I at Harvard. The Mark-I was built by a team of engineers from IBM. He followed the Mark-I with a relay machine, the Mark-II, and a pair of vacuum tube machines, the Mark-III and Mark-IV. The Mark-III and Mark-IV were built after the first stored-program machines. Because they had separate memories for instructions and data, the machines were regarded as reactionary by the advocates of stored-program computers. The term *Harvard architecture* was coined to describe this type of machine. Though clearly different from the original sense, this term is used today to apply to machines with a single main memory but with separate instruction and data caches.

The Whirlwind project [Redmond and Smith 1980] began at MIT in 1947 and was aimed at applications in real-time radar signal processing. Although it led to several inventions, its overwhelming innovation was the creation of magnetic core memory, the first reliable and inexpensive memory technology. Whirlwind had 2048 16-bit words of magnetic core. Magnetic cores served as the main memory technology for nearly 30 years.

Important Special-Purpose Machines

During World War II, major computing efforts in both Great Britain and the United States focused on special-purpose code-breaking computers. The work in Great Britain was aimed at decrypting messages encoded with the German Enigma coding machine. This work, which occurred at a location called Bletchley Park, led to two important machines. The first, an electromechanical machine, conceived of by Alan Turing, was called BOMB [see Good in Metropolis, Howlett, and Rota 1980]. The second, much larger and electronic machine, conceived and designed by Newman and Flowers, was called COLOSSUS [see Randall in Metropolis, Howlett, and Rota 1980]. These were highly specialized cryptanalysis machines, which played a vital role in the war by providing the ability to read coded messages, especially those sent to U-boats. The work at Bletchley Park was highly classified (indeed, some of it is still classified), so its direct impact on the development of ENIAC, EDSAC, and other computers is difficult to trace, but it certainly had an indirect effect in advancing the technology and gaining understanding of the issues.

Similar work on special-purpose computers for cryptanalysis went on in the United States. The most direct descendent of this effort was the company Engineering Research Associates (ERA) [see Thomash in Metropolis, Howlett, and Rota 1980], which was founded after the war to attempt to commercialize on the key ideas. ERA built several machines that were sold to secret government agencies, and it was eventually purchased by Sperry-Rand, which had earlier purchased the Eckert Mauchly Computer Corporation.

Another early set of machines that deserves credit was a group of special-purpose machines built by Konrad Zuse in Germany in the late 1930s and early 1940s [see Bauer and Zuse in Metropolis, Howlett, and Rota 1980]. In addition to producing an operating machine, Zuse was the first to implement floating point, which von Neumann claimed was unnecessary! His early machines used a mechanical store that was smaller than other electromechanical solutions of the time. His last machine was electromechanical but, because of the war, was never completed.

An important early contributor to the development of electronic computers was John Atanasoff, who built a small-scale electronic computer in the early 1940s [Atanasoff 1940]. His machine, designed at Iowa State University, was a special-purpose computer (called the ABC, for Atanasoff Berry Computer) that was never completely operational. Mauchly briefly visited Atanasoff before he built ENIAC, and several of Atanasoff's ideas (e.g., using binary representation) likely influenced Mauchly. The presence of the Atanasoff machine, delays in filing the ENIAC patents (the work was classified, and patents could not be filed until after the war), and the distribution of von Neumann's EDVAC paper were used to break the Eckert–Mauchly patent [Larson 1973]. Though controversy still rages over Atanasoff's role, Eckert and Mauchly are usually given credit for building the first working, general-purpose, electronic computer [Stern 1980]. Atanasoff, however, demonstrated several important innovations included in later computers. Atanasoff deserves much credit for his work, and he might fairly be given credit for the world's first special-purpose electronic computer and for possibly influencing Eckert and Mauchly.

Commercial Developments

In December 1947, Eckert and Mauchly formed Eckert-Mauchly Computer Corporation. Their first machine, the BINAC, was built for Northrop and was shown in August 1949. After some financial difficulties, the Eckert-Mauchly Computer Corporation was acquired by Remington-Rand, later called Sperry-Rand. Sperry-Rand merged the Eckert-Mauchly acquisition, ERA, and its tabulating business to form a dedicated computer division, called UNIVAC. UNIVAC delivered its first computer, the UNIVAC I, in June 1951. The UNIVAC I sold for $250,000 and was the first successful commercial computer—48 systems were built! Today, this early machine, along with many other fascinating pieces of computer lore, can be seen at the Computer History Museum in Mountain View, California. Other places where early computing systems can be visited include the Deutsches Museum in Munich and the Smithsonian Institution in Washington, D.C., as well as numerous online virtual museums.

IBM, which earlier had been in the punched card and office automation business, didn't start building computers until 1950. The first IBM computer, the IBM 701 based on von Neumann's IAS machine, shipped in 1952 and eventually sold 19 units [see Hurd in Metropolis, Howlett, and Rota 1980]. In the early 1950s, many people were pessimistic about the future of computers, believing that the market and opportunities for these "highly specialized" machines were quite limited. Nonetheless, IBM quickly became the most successful computer company. Their focus on reliability and customer- and market-driven strategies were key. Although the 701 and 702 were modest successes, IBM's follow-up machines, the 650, 704, and 705 (delivered in 1954 and 1955) were significant successes, each selling from 132 to 1800 computers.

Several books describing the early days of computing have been written by the pioneers [Goldstine 1972; Wilkes 1985, 1995], as well as Metropolis, Howlett, and Rota [1980], which is a collection of recollections by early pioneers. There are numerous independent histories, often built around the people involved [Slater 1987], as well as a journal, *Annals of the History of Computing*, devoted to the history of computing.

Development of Quantitative Performance Measures: Successes and Failures

In the earliest days of computing, designers set performance goals—ENIAC was to be 1000 times faster than the Harvard Mark-I, and the IBM Stretch (7030) was to be 100 times faster than the fastest machine in existence. What wasn't clear, though, was how this performance was to be measured. In looking back over the years, it is a consistent theme that each generation of computers obsoletes the performance evaluation techniques of the prior generation.

The original measure of performance was time to perform an individual operation, such as addition. Since most instructions took the same execution time, the timing of one gave insight into the others. As the execution times of instructions in a machine became more diverse, however, the time for one operation was no longer useful for comparisons. To take these differences into account, an *instruction mix* was calculated by measuring the relative frequency of instructions in a computer across many programs. The Gibson mix [Gibson 1970] was an early popular instruction mix. Multiplying the time for each instruction times its weight in the mix gave the user the *average instruction execution time*. (If measured in clock cycles, average instruction execution time is the same as average cycles per instruction.) Since instruction sets were similar, this was a more accurate comparison than add times. From average instruction execution time, then, it was only a small step to MIPS (as we have seen, the one is the inverse of the other). MIPS had the virtue of being easy for the layperson to understand.

As CPUs became more sophisticated and relied on memory hierarchies and pipelining, there was no longer a single execution time per instruction; MIPS could not be calculated from the mix and the manual. The next step was benchmarking using kernels and synthetic programs. Curnow and Wichmann [1976] created the Whetstone synthetic program by measuring scientific programs written in Algol 60. This program was converted to FORTRAN and was widely used to characterize scientific program performance. An effort with similar goals to Whetstone, the Livermore FORTRAN Kernels, was made by McMahon [1986] and researchers at Lawrence Livermore Laboratory in an attempt to establish a benchmark for supercomputers. These kernels, however, consisted of loops from real programs.

As it became clear that using MIPS to compare architectures with different instruction sets would not work, a notion of relative MIPS was created. When the VAX-11/780 was ready for announcement in 1977, DEC ran small benchmarks that were also run on an IBM 370/158. IBM marketing referred to the 370/158 as a

1 MIPS computer, and, because the programs ran at the same speed, DEC marketing called the VAX-11/780 a 1 MIPS computer. Relative MIPS for a machine M was defined based on some reference machine as:

$$\text{MIPS}_{\text{M}} = \frac{\text{Performance}_{\text{M}}}{\text{Performance}_{\text{reference}}} \times \text{MIPS}_{\text{reference}}$$

The popularity of the VAX-11/780 made it a popular reference machine for relative MIPS, especially since relative MIPS for a 1 MIPS computer is easy to calculate: If a machine was five times faster than the VAX-11/780, for that benchmark its rating would be 5 relative MIPS. The 1 MIPS rating was unquestioned for 4 years, until Joel Emer of DEC measured the VAX-11/780 under a time-sharing load. He found that the VAX-11/780 native MIPS rating was 0.5. Subsequent VAXes that ran 3 native MIPS for some benchmarks were therefore called 6 MIPS machines because they ran six times faster than the VAX-11/780. By the early 1980s, the term *MIPS* was almost universally used to mean relative MIPS.

The 1970s and 1980s marked the growth of the supercomputer industry, which was defined by high performance on floating-point-intensive programs. Average instruction time and MIPS were clearly inappropriate metrics for this industry, hence the invention of MFLOPS (millions of floating-point operations per second), which effectively measured the inverse of execution time for a benchmark. Unfortunately, customers quickly forget the program used for the rating, and marketing groups decided to start quoting peak MFLOPS in the supercomputer performance wars.

SPEC (System Performance and Evaluation Cooperative) was founded in the late 1980s to try to improve the state of benchmarking and make a more valid basis for comparison. The group initially focused on workstations and servers in the UNIX marketplace, and these remain the primary focus of these benchmarks today. The first release of SPEC benchmarks, now called SPEC89, was a substantial improvement in the use of more realistic benchmarks. SPEC2006 still dominates processor benchmarks almost two decades later.

References

Amdahl, G. M. [1967]. “Validity of the single processor approach to achieving large scale computing capabilities,” *Proc. AFIPS Spring Joint Computer Conf.*, April 18–20, 1967, Atlantic City, N.J., 483–485.

Atanasoff, J. V. [1940]. “Computing machine for the solution of large systems of linear equations,” Internal Report, Iowa State University, Ames.

Azizi, O., Mahesri, A., Lee, B. C., Patel, S. J., & Horowitz, M. [2010]. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. Proc. International Symposium on Computer Architecture, 26-36.

Bell, C. G. [1984]. “The mini and micro industries,” *IEEE Computer* 17:10 (October), 14–30.

Bell, C. G., J. C. Mudge, and J. E. McNamara [1978]. *A DEC View of Computer Engineering*, Digital Press, Bedford, Mass.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," Report to the U.S. Army Ordnance Department, p. 1; also appears in *Papers of John von Neumann*, W. Aspray and A. Burks, eds., MIT Press, Cambridge, Mass., and Tomash Publishers, Los Angeles, Calif., 1987, 97–146.

Curnow, H. J., and B. A. Wichmann [1976]. "A synthetic benchmark," *The Computer J.* 19:1, 43–49.

Dally, William J., "High Performance Hardware for Machine Learning," Cadence Embedded Neural Network Summit, February 9, 2016. http://ip.cadence.com/uploads/presentations/1000AM_Dally_Cadence_ENN.pdf

Flemming, P. J., and J. J. Wallace [1986]. "How not to lie with statistics: The correct way to summarize benchmarks results," *Communications of the ACM* 29:3 (March), 218–221.

Fuller, S. H., and W. E. Burr [1977]. "Measurement and evaluation of alternative computer architectures," *Computer* 10:10 (October), 24–35.

Gibson, J. C. [1970]. "The Gibson mix," Rep. TR. 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y. (research done in 1959).

Goldstine, H. H. [1972]. *The Computer: From Pascal to von Neumann*, Princeton University Press, Princeton, N.J.

Gray, J., and C. van Ingen [2005]. *Empirical Measurements of Disk Failure Rates and Error Rates*, MSR-TR-2005-166, Microsoft Research, Redmond, Wash.

Jain, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.

Kembel, R. [2000]. "Fibre Channel: A comprehensive introduction," *Internet Week* (April).

Larson, E. R. [1973]. "Findings of fact, conclusions of law, and order for judgment," File No. 4-67, Civ. 138, *Honeywell v. Sperry-Rand and Illinois Scientific Development*, U.S. District Court for the State of Minnesota, Fourth Division (October 19).

Lubeck, O., J. Moore, and R. Mendez [1985]. "A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2," *Computer* 18:12 (December), 10–24.

Landstrom, B. [2014]. "The Cost Of Downtime," http://www.interxion.com/blogs/2014/07/the-cost-of-downtime/

McMahon, F. M. [1986]. *The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range*, Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, University of California, Livermore.

Metropolis, N., J. Howlett, and G. C. Rota, eds. [1980]. *A History of Computing in the Twentieth Century*, Academic Press, New York.

Mukherjee S. S., C. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin [2003]. "Measuring architectural vulnerability factors," *IEEE Micro* 23:6, 70–75.

Oliker, L., A. Canning, J. Carter, J. Shalf, and S. Ethier [2004]. "Scientific computations on modern parallel vector systems," *Proc. ACM/IEEE Conf. on Supercomputing*, November 6–12, 2004, Pittsburgh, Penn., 10.

Patterson, D. [2004]. “Latency lags bandwidth,” *Communications of the ACM* 47:10 (October), 71–75.

Redmond, K. C., and T. M. Smith [1980]. *Project Whirlwind—The History of a Pioneer Computer*, Digital Press, Boston.

Shurkin, J. [1984]. *Engines of the Mind: A History of the Computer*, W. W. Norton, New York.

Slater, R. [1987]. *Portraits in Silicon*, MIT Press, Cambridge, Mass.

Smith, J. E. [1988]. “Characterizing computer performance with a single number,” *Communications of the ACM* 31:10 (October), 1202–1206.

SPEC. [1989]. *SPEC Benchmark Suite Release 1.0* (October 2).

SPEC. [1994]. *SPEC Newsletter* (June).

Stern, N. [1980]. “Who invented the first electronic digital computer?” *Annals of the History of Computing* 2:4 (October), 375–376.

Touma, W. R. [1993]. *The Dynamics of the Computer Industry: Modeling the Supply of Workstations and Their Components*, Kluwer Academic, Boston.

Weicker, R. P. [1984]. “Dhrystone: A synthetic systems programming benchmark,” *Communications of the ACM* 27:10 (October), 1013–1030.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, Mass.

Wilkes, M. V. [1995]. *Computing Perspectives*, Morgan Kaufmann, San Francisco.

Wilkes, M. V., D. J. Wheeler, and S. Gill [1951]. *The Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley, Cambridge, Mass.

M.3

The Development of Memory Hierarchy and Protection (Chapter 2 and Appendix B)

Although the pioneers of computing knew of the need for a memory hierarchy and coined the term, the automatic management of two levels was first proposed by Kilburn et al. [1962]. It was demonstrated with the Atlas computer at the University of Manchester. This computer appeared the year before the IBM 360 was announced. Although IBM planned for its introduction with the next generation (System/370), the operating system TSS was not up to the challenge in 1970. Virtual memory was announced for the 370 family in 1972, and it was for this computer that the term *translation lookaside buffer* was coined [Case and Padegs 1978]. The only computers today without virtual memory are a few supercomputers, embedded processors, and older personal computers.

Both the Atlas and the IBM 360 provided protection on pages, and the GE 645 was the first system to provide paged segmentation. The earlier Burroughs computers provided virtual memory using segmentation, similar to the segmented address scheme of the Intel 8086. The 80286, the first 80x86 to have the protection mechanisms described in Appendix C, was inspired by the Multics protection software that ran on the GE 645. Over time, computers evolved more elaborate mechanisms. The most elaborate mechanism was *capabilities*, which attracted the greatest interest in the late 1970s and early 1980s [Fabry 1974; Wulf, Levin, and Harbison 1981]. Wilkes [1982], one of the early workers on capabilities, had this to say:

> *Anyone who has been concerned with an implementation of the type just described [capability system], or has tried to explain one to others, is likely to feel that complexity has got out of hand. It is particularly disappointing that the attractive idea of capabilities being tickets that can be freely handed around has become lost ….*
>
> *Compared with a conventional computer system, there will inevitably be a cost to be met in providing a system in which the domains of protection are small and frequently changed. This cost will manifest itself in terms of additional hardware, decreased runtime speed, and increased memory occupancy. It is at present an open question whether, by adoption of the capability approach, the cost can be reduced to reasonable proportions. [p. 112]*

Today there is little interest in capabilities either from the operating systems or the computer architecture communities, despite growing interest in protection and security.

Bell and Strecker [1976] reflected on the PDP-11 and identified a small address space as the only architectural mistake that is difficult to recover from. At the time of the creation of PDP-11, core memories were increasing at a very slow rate. In addition, competition from 100 other minicomputer companies meant that DEC might not have a cost-competitive product if every address had to go through the 16-bit data path twice, hence the architect's decision to add only 4 more address bits than found in the predecessor of the PDP-11.

The architects of the IBM 360 were aware of the importance of address size and planned for the architecture to extend to 32 bits of address. Only 24 bits were used in the IBM 360, however, because the low-end 360 models would have been even slower with the larger addresses in 1964. Unfortunately, the architects didn't reveal their plans to the software people, and programmers who stored extra information in the upper 8 "unused" address bits foiled the expansion effort. (Apple made a similar mistake 20 years later with the 24-bit address in the Motorola 68000, which required a procedure to later determine "32-bit clean" programs for the Macintosh when later 68000s used the full 32-bit virtual address.) Virtually every computer since then will check to make sure the unused bits stay unused and trap if the bits have the wrong value.

As mentioned in the text, system virtual machines were pioneered at IBM as part of its investigation into virtual memory. IBM's first computer with virtual memory was the IBM 360/67, introduced in 1967. IBM researchers wrote the program CP-67 that created the illusion of several independent 360 computers. They then wrote an interactive, single-user operating system called CMS that ran on these virtual machines. CP-67 led to the product VM/370, and today IBM sells z/VM for its mainframe computers [Meyer and Seawright 1970; Van Vleck 2005].

A few years after the Atlas paper, Wilkes published the first paper describing the concept of a cache [1965]:

> *The use is discussed of a fast core memory of, say, 32,000 words as slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory. [p. 270]*

This two-page paper describes a direct-mapped cache. Although this is the first publication on caches, the first implementation was probably a direct-mapped instruction cache built at the University of Cambridge. It was based on tunnel diode memory, the fastest form of memory available at the time. Wilkes stated that G. Scarott suggested the idea of a cache memory.

Subsequent to that publication, IBM started a project that led to the first commercial computer with a cache, the IBM 360/85 [Liptay 1968]. Gibson [1967] described how to measure program behavior as memory traffic as well as miss rate and showed how the miss rate varies between programs. Using a sample of 20 programs (each with 3 million references!), Gibson also relied on average memory access time to compare systems with and without caches. This precedent is more than 40 years old, and yet many used miss rates until the early 1990s.

Conti, Gibson, and Pitkowsky [1968] described the resulting performance of the 360/85. The 360/91 outperforms the 360/85 on only 3 of the 11 programs in the paper, even though the 360/85 has a slower clock cycle time (80 ns versus 60 ns), less memory interleaving (4 versus 16), and a slower main memory (1.04 microsecond versus 0.75 microsecond). This paper was also the first to use the term *cache*.

Others soon expanded the cache literature. Strecker [1976] published the first comparative cache design paper examining caches for the PDP-11. Smith [1982] later published a thorough survey paper that used the terms *spatial locality* and *temporal locality*; this paper has served as a reference for many computer designers.

Although most studies relied on simulations, Clark [1983] used a hardware monitor to record cache misses of the VAX-11/780 over several days. Clark and Emer [1985] later compared simulations and hardware measurements for translations.

Hill [1987] proposed the three C's used in Appendix B to explain cache misses. Jouppi [1998] retrospectively said that Hill's three C's model led directly to his invention of the victim cache to take advantage of faster direct-mapped caches and yet avoid most of the cost of conflict misses. Sugumar and Abraham [1993] argued that the baseline cache for the three C's model should use optimal replacement; this would eliminate the anomalies of least recently used (LRU)-based miss classification and allow conflict misses to be broken down into those caused by mapping and those caused by a nonoptimal replacement algorithm.

One of the first papers on nonblocking caches was by Kroft [1981]. Kroft [1998] later explained that he was the first to design a computer with a cache at

Control Data Corporation, and when using old concepts for new mechanisms he hit upon the idea of allowing his two-ported cache to continue to service other accesses on a miss.

Baer and Wang [1988] did one of the first examinations of the multilevel inclusion property. Wang, Baer, and Levy [1989] then produced an early paper on performance evaluation of multilevel caches. Later, Jouppi and Wilton [1994] proposed multilevel exclusion for multilevel caches on chip.

In addition to victim caches, Jouppi [1990] also examined prefetching via streaming buffers. His work was extended by Farkas, Jouppi, and Chow [1995] to streaming buffers that work well with nonblocking loads and speculative execution for in-order processors, and later Farkas et al. [1997] showed that, while out-of-order processors can tolerate unpredictable latency better, they still benefit. They also refined memory bandwidth demands of stream buffers.

Proceedings of the Symposium on Architectural Support for Compilers and Operating Systems (ASPLOS) and the International Computer Architecture Symposium (ISCA) from the 1990s are filled with papers on caches. (In fact, some wags claimed ISCA really stood for the International *Cache* Architecture Symposium.)

Chapter 2 relies on the measurements of SPEC2000 benchmarks collected by Cantin and Hill [2001]. There are several other papers used in Chapter 2 that are cited in the captions of the figures that use the data: Agarwal and Pudar [1993]; Barroso, Gharachorloo, and Bugnion [1998]; Farkas and Jouppi [1994]; Jouppi [1990]; Lam, Rothberg, and Wolf [1991]; Lebeck and Wood [1994]; McCalpin [2005]; Mowry, Lam, and Gupta [1992]; and Torrellas, Gupta, and Hennessy [1992].

References

Agarwal, A. [1987]. "Analysis of Cache Performance for Operating Systems and Multiprogramming," Ph.D. thesis, Tech. Rep. No. CSL-TR-87-332, Stanford University, Palo Alto, Calif.

Agarwal, A., and S. D. Pudar [1993]. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," *20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif. (*Computer Architecture News* 21:2 (May), 179–190).

Baer, J.-L., and W.-H. Wang [1988]. "On the inclusion property for multi-level cache hierarchies," *Proc. 15th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 30–June 2, 1988, Honolulu, Hawaii, 73–80.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, and R. Neugebauer [2003]. "Xen and the art of virtualization," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, October 19–22, 2003, Bolton Landing, N.Y.

Barroso, L. A., K. Gharachorloo, and E. Bugnion [1998]. "Memory system characterization of commercial workloads," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 3–14.

Bell, C. G., and W. D. Strecker [1976]. "Computer structures: What have we learned from the PDP-11?" *Proc. Third Annual Int'l. Symposium on Computer Architecture (ISCA)*, January 19–21, 1976, Tampa, Fla., 1–14.

Bhandarkar, D. P. [1995]. *Alpha Architecture Implementations*, Digital Press, Newton, Mass.

Borg, A., R. E. Kessler, and D. W. Wall [1990]. "Generation and analysis of very long address traces," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 270–279.

Cantin, J. F., and M. D. Hill [2001]. "Cache performance for selected SPEC CPU2000 benchmarks," *http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/*.

Cantin, J., and M. Hill [2003]. "Cache performance for SPEC CPU2000 benchmarks, version 3.0," *http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/index.html*.

Case, R. P., and A. Padegs [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 830–855.

Clark, B., T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Neefe Matthews [2004]. "Xen and the art of repeated research," *Proc. USENIX Annual Technical Conf.*, June 27–July 2, 2004, Boston, 1135–1144.

Clark, D. W. [1983]. "Cache performance of the VAX-11/780," *ACM Trans. on Computer Systems* 1:1, 24–37.

Clark, D. W., and J. S. Emer [1985]. "Performance of the VAX-11/780 translation buffer: Simulation and measurement," *ACM Trans. on Computer Systems* 3:1 (February), 31–62.

Compaq Computer Corporation. [1999]. *Compiler Writer's Guide for the Alpha 21264*, Order Number EC-RJ66A-TE, June.

Conti, C., D. H. Gibson, and S. H. Pitkowsky [1968]. "Structural aspects of the System/360 Model 85. Part I. General organization," *IBM Systems J.* 7:1, 2–14.

Crawford, J., and P. Gelsinger [1988]. *Programming the 80386*, Sybex, Alameda, Calif.

Cvetanovic, Z., and R. E. Kessler [2000]. "Performance analysis of the Alpha 21264-based Compaq ES40 system," *Proc. 27th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 10–14, 2000, Vancouver, Canada, 192–202.

Fabry, R. S. [1974]. "Capability based addressing," *Communications of the ACM* 17:7 (July), 403–412.

Farkas, K. I., P. Chow, N. P. Jouppi, and Z. Vranesic [1997]. "Memory-system design considerations for dynamically-scheduled processors," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 133–143.

Farkas, K. I., and N. P. Jouppi [1994]. "Complexity/performance trade-offs with non-blocking loads," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago.

Farkas, K. I., N. P. Jouppi, and P. Chow [1995]. "How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors?" *Proc. First IEEE Symposium on High-Performance Computer Architecture*, January 22–25, 1995, Raleigh, N.C., 78–89.

Gao, Q. S. [1993]. "The Chinese remainder theorem and the prime memory system," *20th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif. (*Computer Architecture News* 21:2 (May), 337–340).

Gee, J. D., M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith [1993]. "Cache performance of the SPEC92 benchmark suite," *IEEE Micro* 13:4 (August), 17–27.

Gibson, D. H. [1967]. "Considerations in block-oriented systems design," *AFIPS Conf. Proc.* 30, 75–80.

Handy, J. [1993]. *The Cache Memory Book*, Academic Press, Boston.

Heald, R., K. Aingaran, C. Amir, M. Ang, M. Boland, A. Das, P. Dixit, G. Gouldsberry, J. Hart, T. Horel, W.-J. Hsu, J. Kaku, C. Kim, S. Kim, F. Klass, H. Kwan, R. Lo, H. McIntyre, A. Mehta, D. Murata, S. Nguyen, Y.-P. Pai, S. Patel, K. Shin, K. Tam, S. Vishwanthaiah, J. Wu, G. Yee, and H. You [2000]. "Implementation of third-generation SPARC V9 64-b microprocessor," *ISSCC Digest of Technical Papers*, 412–413 and slide supplement.

Hill, M. D. [1987]. "Aspects of Cache Memory and Instruction Buffer Performance," Ph.D. thesis, Tech. Rep. UCB/CSD 87/381, Computer Science Division, University of California, Berkeley.

Hill, M. D. [1988]. "A case for direct mapped caches," *Computer* 21:12 (December), 25–40.

Horel, T., and G. Lauterbach [1999]. "UltraSPARC-III: Designing third-generation 64-bit performance," *IEEE Micro* 19:3 (May–June), 73–85.

Hughes, C. J., P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan [2001]. "Variability in the execution of multimedia applications and implications for architecture," *Proc. 28th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 30–July 4, 2001, Goteborg, Sweden, 254–265.

IEEE. [2005]. "Intel virtualization technology, computer," *IEEE Computer Society* 38:5 (May), 48–56.

Jouppi, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 364–373.

Jouppi, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in G. S. Sohi, ed., *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 71–73.

Jouppi, N. P., and S. J. E. Wilton [1994]. "Trade-offs in two-level on-chip caching," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 34–45.

Kessler, R. E. [1999]. "The Alpha 21264 microprocessor," *IEEE Micro* 19:2 (March/April), 24–36.

Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner [1962]. "One-level storage system," *IRE Trans. on Electronic Computers* EC-11 (April) 223–235. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 135–148.

Kroft, D. [1981]. "Lockup-free instruction fetch/prefetch cache organization," *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 12–14, 1981, Minneapolis, Minn., 81–87.

Kroft, D. [1998]. "Retrospective: Lockup-free instruction fetch/prefetch cache organization," in G. S. Sohi, ed., *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 20–21.

Kunimatsu, A., N. Ide, T. Sato, Y. Endo, H. Murakami, T. Kamei, M. Hirano, F. Ishihara, H. Tago, M. Oka, A. Ohba, T. Yutaka, T. Okada, and M. Suzuoki [2000]. "Vector unit architecture for emotion synthesis," *IEEE Micro* 20:2 (March–April), 40–47.

Lam, M. S., E. E. Rothberg, and M. E. Wolf [1991]. "The cache performance and optimizations of blocked algorithms," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Santa Clara, Calif. (*SIGPLAN Notices* 26:4 (April), 63–74).

Lebeck, A. R., and D. A. Wood [1994]. "Cache profiling and the SPEC benchmarks: A case study," *Computer* 27:10 (October), 15–26.

Liptay, J. S. [1968]. "Structural aspects of the System/360 Model 85. Part II. The cache," *IBM Systems J.* 7:1, 15–21.

Luk, C.-K., and T. C Mowry [1999]. "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Trans. on Computers*, 48:2 (February), 134–141.

McCalpin, J. D. [2005]. "STREAM: Sustainable Memory Bandwidth in High Performance Computers," *www.cs.virginia.edu/stream/*.

McFarling, S. [1989]. "Program optimization for instruction caches," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 183–191.

Menon, A., J. Renato Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel [2005]. "Diagnosing performance overheads in the xen virtual machine environment," *Proc. First ACM/USENIX Int'l. Conf. on Virtual Execution Environments*, June 11–12, 2005, Chicago, 13–23.

Meyer, R. A., and L. H. Seawright [1970]. "A virtual machine time sharing system," *IBM Systems J.* 9:3, 199–218.

Mowry, T. C., S. Lam, and A. Gupta [1992]. "Design and evaluation of a compiler algorithm for prefetching," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston (*SIGPLAN Notices* 27:9 (September), 62–73).

Oka, M., and M. Suzuoki [1999]. "Designing and programming the emotion engine," *IEEE Micro* 19:6 (November–December), 20–28.

Pabst, T. [2000]. "Performance Showdown at 133 MHz FSB—The Best Platform for Coppermine," *www6.tomshardware.com/mainboard/00q1/000302/*.

Palacharla, S., and R. E. Kessler [1994]. "Evaluating stream buffers as a secondary cache replacement," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 24–33.

Przybylski, S. A. [1990]. *Cache Design: A Performance-Directed Approach*, Morgan Kaufmann, San Francisco.

Przybylski, S. A., M. Horowitz, and J. L. Hennessy [1988]. "Performance tradeoffs in cache design," *Proc. 15th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 30–June 2, 1988, Honolulu, Hawaii, 290–298.

Reinman, G., and N. P. Jouppi. [1999]. "Extensions to CACTI."

Robin, J., and C. Irvine [2000]. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," *Proc. USENIX Security Symposium*, August 14–17, 2000, Denver, Colo.

Saavedra-Barrera, R. H. [1992]. "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking," Ph.D. dissertation, University of California, Berkeley.

Samples, A. D., and P. N. Hilfinger [1988]. *Code Reorganization for Instruction Caches*, Tech. Rep. UCB/CSD 88/447, University of California, Berkeley.

Sites, R. L. (ed.) [1992]. *Alpha Architecture Reference Manual*, Digital Press, Burlington, Mass.

Skadron, K., and D. W. Clark [1997]. "Design issues and tradeoffs for write buffers," *Proc. Third Int'l. Symposium on High-Performance Computer Architecture*, February 1–5, 1997, San Antonio, Tex., 144–155.

Smith, A. J. [1982]. "Cache memories," *Computing Surveys* 14:3 (September), 473–530.

Smith, J. E., and J. R. Goodman [1983]. "A study of instruction cache organizations and replacement policies," *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 132–137.

Stokes, J. [2000]. "Sound and Vision: A Technical Overview of the Emotion Engine," *http://arstechnica.com/hardware/reviews/2000/02/ee.ars*.

Strecker, W. D. [1976]. "Cache memories for the PDP-11?" *Proc. Third Annual Int'l. Symposium on Computer Architecture (ISCA)*, January 19–21, 1976, Tampa, Fla., 155–158.

Sugumar, R. A., and S. G. Abraham [1993]. "Efficient simulation of caches under optimal replacement with applications to miss characterization," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 17–21, 1993, Santa Clara, Calif., 24–35.

Tarjan, D., S. Thoziyoor, and N. Jouppi [2006]. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories.

Torrellas, J., A. Gupta, and J. Hennessy [1992]. "Characterizing the caching and synchronization performance of a multiprocessor operating system," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston (*SIGPLAN Notices* 27:9 (September), 162–174).

Van Vleck, T. [2005]. "The IBM 360/67 and CP/CMS," *http://www.multicians.org/thvv/360-67.html*.

Wang, W.-H., J.-L. Baer, and H. M. Levy [1989]. "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–June 1, 1989, Jerusalem, 140–148.

Wilkes, M. [1965]. "Slave memories and dynamic storage allocation," *IEEE Trans. Electronic Computers* EC-14:2 (April), 270–271.

Wilkes, M. V. [1982]. "Hardware support for memory protection: Capability implementations," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 107–116.

Wulf, W. A., R. Levin, and S. P. Harbison [1981]. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York.

M.4 The Evolution of Instruction Sets (Appendices A, J, and K)

> *One's eyebrows should rise whenever a future architecture is developed with a stack- or register-oriented instruction set.*
>
> **Meyers [1978, p. 20]**

The earliest computers, including the UNIVAC I, the EDSAC, and the IAS computers, were accumulator-based computers. The simplicity of this type of computer made it the natural choice when hardware resources were very constrained. The first general-purpose register computer was the Pegasus, built by Ferranti, Ltd., in 1956. The Pegasus had eight general-purpose registers, with R0 always being zero. Block transfers loaded the eight registers from the drum memory.

Stack Architectures

In 1963, Burroughs delivered the B5000. The B5000 was perhaps the first computer to seriously consider software and hardware-software trade-offs. Barton and the designers at Burroughs made the B5000 a stack architecture (as described in Barton [1961]). Designed to support high-level languages such as ALGOL, this stack architecture used an operating system (MCP) written in a high-level language. The B5000 was also the first computer from a U.S. manufacturer to support virtual memory. The B6500, introduced in 1968 (and discussed in Hauck and Dent [1968]), added hardware-managed activation records. In both the B5000 and B6500, the top two elements of the stack were kept in the processor and the rest of the stack was kept in memory. The stack architecture yielded good code density, but only provided two high-speed storage locations. The authors of both the original IBM 360 paper [Amdahl, Blaauw, and Brooks 1964] and the original PDP-11 paper [Bell et al. 1970] argued against the stack organization. They cited three major points in their arguments against stacks:

- Performance is derived from fast registers, not the way they are used.
- The stack organization is too limiting and requires many swap and copy operations.
- The stack has a bottom, and when placed in slower memory there is a performance loss.

Stack-based hardware fell out of favor in the late 1970s and, except for the Intel 80x86 floating-point architecture, essentially disappeared; for example, except for the 80x86, none of the computers listed in the SPEC report uses a stack.

In the 1990s, however, stack architectures received a shot in the arm with the success of the Java Virtual Machine (JVM). The JVM is a software interpreter for an intermediate language produced by Java compilers, called *Java bytecodes* [Lindholm and Yellin 1999]. The purpose of the interpreter is to provide software compatibility across many platforms, with the hope of "write once, run everywhere." Although the slowdown is about a factor of 10 due to interpretation, there are times when compatibility is more important than performance, such as when downloading a Java "applet" into an Internet browser.

Although a few have proposed hardware to directly execute the JVM instructions (see McGhan and O'Connor [1998]), thus far none of these proposals has been significant commercially. The hope instead is that *just-in-time* (JIT) Java compilers—which compile during runtime to the native instruction set of the computer running the Java program—will overcome the performance penalty of interpretation. The popularity of Java has also led to compilers that compile directly into the native hardware instruction sets, bypassing the illusion of the Java bytecodes.

Computer Architecture Defined

IBM coined the term *computer architecture* in the early 1960s. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the IBM 360 instruction set. They believed that a *family* of computers of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at that time. IBM, although it was the leading company in the industry, had five different architectures before the 360; thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that defining a common architecture would bring six different divisions of IBM together. Their definition of architecture was

> *… the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.*

The term *machine language programmer* meant that compatibility would hold, even in machine language, while *timing independent* allowed different implementations. This architecture blazed the path for binary compatibility, which others have followed.

The IBM 360 was the first computer to sell in large quantities with both byte addressing using 8-bit bytes and general-purpose registers. The 360 also had register-memory and limited memory-memory instructions. Appendix K summarizes this instruction set.

In 1964, Control Data delivered the first supercomputer, the CDC 6600. As Thornton [1964] discussed, he, Cray, and the other 6600 designers were among the first to explore pipelining in depth. The 6600 was the first general-purpose, load-store computer. In the 1960s, the designers of the 6600 realized the need to simplify architecture for the sake of efficient pipelining. Microprocessor and minicomputer designers largely neglected this interaction between architectural simplicity and implementation during the 1970s, but it returned in the 1980s.

High-Level Language Computer Architecture

In the late 1960s and early 1970s, people realized that software costs were growing faster than hardware costs. McKeeman [1967] argued that compilers and operating systems were getting too big and too complex and taking too long to develop. Because of inferior compilers and the memory limitations of computers, most systems programs at the time were still written in assembly language. Many researchers proposed alleviating the software crisis by creating more powerful, software-oriented architectures. Tanenbaum [1978] studied the properties of high-level languages. Like other researchers, he found that most programs are simple. He argued that architectures should be designed with this in mind and that they should optimize for program size and ease of compilation. Tanenbaum proposed a stack computer with frequency-encoded instruction formats to accomplish these goals; however, as we have observed, program size does not translate directly to cost-performance, and stack computers faded out shortly after this work.

Strecker's article [1978] discusses how he and the other architects at DEC responded to this by designing the VAX architecture. The VAX was designed to simplify compilation of high-level languages. Compiler writers had complained about the lack of complete orthogonality in the PDP-11. The VAX architecture was designed to be highly orthogonal and to allow the mapping of a high-level language statement into a single VAX instruction. Additionally, the VAX designers tried to optimize code size because compiled programs were often too large for available memories. Appendix K summarizes this instruction set.

The VAX-11/780 was the first computer announced in the VAX series. It is one of the most successful—and most heavily studied—computers ever built. The cornerstone of DEC's strategy was a single architecture, VAX, running a single operating system, VMS. This strategy worked well for over 10 years. The large number of papers reporting instruction mixes, implementation measurements, and analysis of the VAX makes it an ideal case study [Clark and Levy 1982; Wiecek 1982]. Bhandarkar and Clark [1991] gave a quantitative analysis of the disadvantages of the VAX versus a RISC computer, essentially a technical explanation for the demise of the VAX.

While the VAX was being designed, a more radical approach, called *high-level language computer architecture* (HLLCA), was being advocated in the research community. This movement aimed to eliminate the gap between high-level languages and computer hardware—what Gagliardi [1973] called the "semantic gap"—by bringing the hardware "up to" the level of the programming language. Meyers [1982] provided a good summary of the arguments and a history of high-level language computer architecture projects. HLLCA never had a significant commercial impact. The increase in memory size on computers eliminated the code size problems arising from high-level languages and enabled operating systems to be written in high-level languages. The combination of simpler architectures together with software offered greater performance and more flexibility at lower cost and lower complexity.

Reduced Instruction Set Computers

In the early 1980s, the direction of computer architecture began to swing away from providing high-level hardware support for languages. Ditzel and Patterson [1980] analyzed the difficulties encountered by the high-level language architectures and argued that the answer lay in simpler architectures. In another paper [Patterson and Ditzel 1980], these authors first discussed the idea of Reduced Instruction Set Computers (RISCs) and presented the argument for simpler architectures. Clark and Strecker [1980], who were VAX architects, rebutted their proposal.

The simple load-store computers such as MIPS are commonly called RISC architectures. The roots of RISC architectures go back to computers like the 6600, where Thornton, Cray, and others recognized the importance of instruction set simplicity in building a fast computer. Cray continued his tradition of keeping computers simple in the CRAY-1. Commercial RISCs are built primarily on the work of three research projects: the Berkeley RISC processor, the IBM 801, and the Stanford MIPS processor. These architectures have attracted enormous industrial interest because of claims of a performance advantage of anywhere from two to five times over other computers using the same technology.

Begun in 1975, the IBM project was the first to start but was the last to become public. The IBM computer was designed as a 24-bit ECL minicomputer, while the university projects were both MOS-based, 32-bit microprocessors. John Cocke is considered the father of the 801 design. He received both the Eckert–Mauchly and Turing awards in recognition of his contribution. Radin [1982] described the highlights of the 801 architecture. The 801 was an experimental project that was never designed to be a product. In fact, to keep down costs and complexity, the computer was built with only 24-bit registers.

In 1980, Patterson and his colleagues at Berkeley began the project that was to give this architectural approach its name (see Patterson and Ditzel [1980]). They built two computers called RISC-I and RISC-II. Because the IBM project was not widely known or discussed, the role played by the Berkeley group in promoting the RISC approach was critical to acceptance of the technology. They also built one of the first instruction caches to support hybrid-format RISCs (see Patterson et al. [1983]). It supported 16-bit and 32-bit instructions in memory but 32 bits in the cache. The Berkeley group went on to build RISC computers targeted toward Smalltalk, described by Ungar et al. [1984], and LISP, described by Taylor et al. [1986].

In 1981, Hennessy and his colleagues at Stanford published a description of the Stanford MIPS computer. Efficient pipelining and compiler-assisted scheduling of the pipeline were both important aspects of the original MIPS design. MIPS stood for Microprocessor without Interlocked Pipeline Stages, reflecting the lack of hardware to stall the pipeline, as the compiler would handle dependencies.

These early RISC computers—the 801, RISC-II, and MIPS—had much in common. Both university projects were interested in designing a simple computer that could be built in VLSI within the university environment. All three computers used a simple load-store architecture and fixed-format 32-bit instructions, and emphasized efficient pipelining. Patterson [1985] described the three computers and the basic design principles that have come to characterize what a RISC computer is, and Hennessy [1984] provided another view of the same ideas, as well as other issues in VLSI processor design.

In 1985, Hennessy published an explanation of the RISC performance advantage and traced its roots to a substantially lower CPI—under 2 for a RISC processor and over 10 for a VAX-11/780 (though not with identical workloads). A paper by Emer and Clark [1984] characterizing VAX-11/780 performance was instrumental in helping the RISC researchers understand the source of the performance advantage seen by their computers.

Since the university projects finished up, in the 1983–1984 time frame, the technology has been widely embraced by industry. Many manufacturers of the early computers (those made before 1986) claimed that their products were RISC computers. These claims, however, were often born more of marketing ambition than of engineering reality.

In 1986, the computer industry began to announce processors based on the technology explored by the three RISC research projects. Moussouris et al. [1986] described the MIPS R2000 integer processor, while Kane's book [1986] provides a complete description of the architecture. Hewlett-Packard converted their existing minicomputer line to RISC architectures; Lee [1989] described the HP Precision Architecture. IBM never directly turned the 801 into a product. Instead, the ideas were adopted for a new, low-end architecture that was incorporated in the IBM RT-PC and described in a collection of papers [Waters 1986]. In 1990, IBM announced a new RISC architecture (the RS 6000), which is the first superscalar RISC processor. In 1987, Sun Microsystems began delivering computers based on the SPARC architecture, a derivative of the Berkeley RISC-II processor; SPARC is described in Garner et al. [1988]. The PowerPC joined the forces of Apple, IBM, and Motorola. Appendix K summarizes several RISC architectures.

To help resolve the RISC versus traditional design debate, designers of VAX processors later performed a quantitative comparison of VAX and a RISC processor for implementations with comparable organizations. Their choices were the

VAX 8700 and the MIPS M2000. The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to powerful addressing modes, powerful instructions, efficient instruction encoding, and few registers. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. These goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

Figure M.1 shows the ratio of the number of instructions executed, the ratio of CPIs, and the ratio of performance measured in clock cycles. Since the organizations were similar, clock cycle times were assumed to be the same. MIPS executes about twice as many instructions as the VAX, while the CPI for the VAX is about six times larger than that for the MIPS. Hence, the MIPS M2000 has almost three times the performance of the VAX 8700. Furthermore, much less hardware is needed to build the MIPS processor than the VAX processor. This cost-performance gap is the reason why the company that used to make the VAX introduced a MIPS-based product and then has dropped the VAX completely and switched to Alpha, which is quite similar to MIPS. Bell and Strecker [1998] summarized the debate inside the company. Today, DEC, once the second largest computer company and the major success of the minicomputer industry, exists only as remnants within HP and Intel.

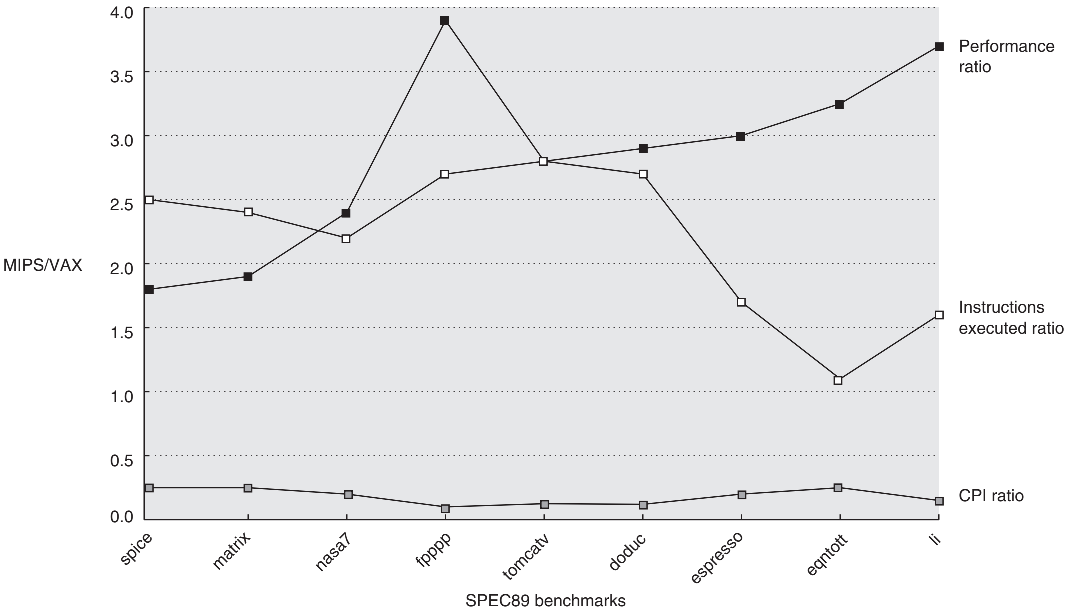


Figure M.1 Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from Bhandarkar and Clark [1991].)

Looking back, only one Complex Instruction Set Computer (CISC) instruction set survived the RISC/CISC debate, and that one had binary compatibility with PC software. The volume of chips is so high in the PC industry that there is a sufficient revenue stream to pay the extra design costs—and sufficient resources due to Moore's law—to build microprocessors that translate from CISC to RISC internally. Whatever loss in efficiency occurred (due to longer pipeline stages and bigger die size to accommodate translation on the chip) was overcome by the enormous volume and the ability to dedicate IC processing lines specifically to this product.

Interestingly, Intel also concluded that the future of the 80x86 line was doubtful. They created the IA-64 architecture to support 64-bit addressing and to move to a RISC-style instruction set. The embodiment of the IA-64 (see Huck et al. [2000]) architecture in the Itanium-1 and Itanium-2 has been a mixed success. Although high performance has been achieved for floating-point applications, the integer performance was never impressive. In addition, the Itanium implementations have been large in transistor count and die size and power hungry. The complexity of the IA-64 instruction set, standing at least in partial conflict with the RISC philosophy, no doubt contributed to this area and power inefficiency.

AMD decided instead to just stretch the architecture from a 32-bit address to a 64-bit address, much as Intel had done when the 80386 stretched it from a 16-bit address to a 32-bit address. Intel later followed AMD's example. In the end, the tremendous marketplace advantage of the 80x86 presence was too much even for Intel, the owner of this legacy, to overcome!

References

Alexander, W. G., and D. B. Wortman [1975]. "Static and dynamic characteristics of XPL programs," *IEEE Computer* 8:11 (November), 41–46.

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. [1964]. "Architecture of the IBM System 360," *IBM J. Research and Development* 8:2 (April), 87–101.

Barton, R. S. [1961]. "A new approach to the functional design of a computer," *Proc. Western Joint Computer Conf.*, May 9–11, 1961, Los Angeles, Calif., 393–396.

Bell, G., R. Cady, H. McFarland, B. DeLagi, J. O'Laughlin, R. Noonan, and W. Wulf [1970]. "A new architecture for mini-computers: The DEC PDP-11," *Proc. AFIPS SJCC*, May 5–7, 1970, Atlantic City, N.J., 657–675.

Bell, G., and W. D. Strecker [1998]. "Computer structures: What have we learned from the PDP-11?" in G. S. Sohi, ed., *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ACM, New York, 138–151.

Bhandarkar, D. P. [1995]. *Alpha Architecture and Implementations*, Digital Press, Newton, Mass.

Bhandarkar, D., and D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth*

Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 8–11, 1991, Palo Alto, Calif., 310–319.

Bier, J. [1997]. "The evolution of DSP processors," paper presented at University of California, Berkeley, November 14.

Boddie, J. R. [2000]. "History of DSPs," *www.lucent.com/micro/dsp/dsphist.html*.

Case, R. P., and A. Padegs [1978]. "The architecture of the IBM System/370," *Communications of the ACM* 21:1, 73–96.

Chow, F. C. [1983]. "A Portable Machine-Independent Global Optimizer—Design and Measurements," Ph.D. thesis, Stanford University, Palo Alto, Calif.

Clark, D., and H. Levy [1982]. "Measurement and analysis of instruction set use in the VAX-11/780," *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 26–29, 1982, Austin, Tex., 9–17.

Clark, D., and W. D. Strecker [1980]. "Comments on 'the case for the reduced instruction set computer,'" *Computer Architecture News* 8:6 (October), 34–38.

Crawford, J., and P. Gelsinger [1988]. *Programming the 80386*, Sybex Books, Alameda, Calif.

Darcy, J. D., and D. Gay [1996]. "FLECKmarks: Measuring floating point performance using a full IEEE compliant arithmetic benchmark," CS 252 class project, University of California, Berkeley (see *http://www.sonic.net/~jddarcy/Research/fleckmrk.pdf*).

Digital Semiconductor. [1996]. *Alpha Architecture Handbook, Version 3*, Digital Press, Maynard, Mass.

Ditzel, D. R., and D. A. Patterson [1980]. "Retrospective on high-level language computer architecture," *Proc. Seventh Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 6–8, 1980, La Baule, France, 97–104.

Emer, J. S., and D. W. Clark [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 301–310.

Furber, S. B. [2000]. *ARM system-on-chip architecture*. Addison-Wesley, Boston, Mass.

Gagliardi, U. O. [1973]. "Report of workshop 4—software-related advances in computer hardware," *Proc. Symposium on the High Cost of Software*, September 17–19, 1973, Monterey, Calif., 99–120.

Game, M., and A. Booker [1999]. "CodePack code compression for PowerPC processors," *MicroNews*, 5:1.

Garner, R., A. Agarwal, F. Briggs, E. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck [1988]. "Scalable processor architecture (SPARC)," *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 278–283.

Hauck, E. A., and B. A. Dent [1968]. "Burroughs' B6500/B7500 stack mechanism," *Proc. AFIPS SJCC*, April 30–May 2, 1968, Atlantic City, N.J., 245–251.

Hennessy, J. [1984]. "VLSI processor architecture," *IEEE Trans. on Computers* C-33:11 (December), 1221–1246.

Hennessy, J. [1985]. "VLSI RISC processors," *VLSI Systems Design* 6:10 (October), 22–32.

Hennessy, J., N. Jouppi, F. Baskett, and J. Gill [1981]. "MIPS: A VLSI processor architecture," in *CMU Conference on VLSI Systems and Computations*, Computer Science Press, Rockville, Md.

Hewlett-Packard. [1994]. *PA-RISC 2.0 Architecture Reference Manual*, 3rd ed., Hewlett-Packard, Palo Alto, Calif.

Hitachi. [1997]. *SuperH RISC Engine SH7700 Series Programming Manual*, Hitachi, Santa Clara, Calif.

Huck, J. et al. [2000]. "Introducing the IA-64 Architecture" *IEEE Micro*, 20:5, (September–October), 12–23.

IBM. [1994]. *The PowerPC Architecture*, Morgan Kaufmann, San Francisco.

Intel. [2001]. "Using MMX instructions to convert RGB to YUV color conversion," *cedar.intel.com/cgi-bin/ids.dll/content/content.jsp?cntKey=Legacy::irtm_AP548_9996&cntType=IDS_EDITORIAL*.

Kahan, J. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California, Berkeley.

Kane, G. [1986]. *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Kane, G. [1996]. *PA-RISC 2.0 Architecture*, Prentice Hall, Upper Saddle River, N.J.

Kane, G., and J. Heinrich [1992]. *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, N.J.

Kissell, K. D. [1997]. "MIPS16: High-density for the embedded market," *Proc. Real Time Systems '97*, June 15, 1997, Las Vegas, Nev.

Kozyrakis, C. [2000]. "Vector IRAM: A media-oriented vector processor with embedded DRAM," paper presented at Hot Chips 12, August 13–15, 2000, Palo Alto, Calif, 13–15.

Lee, R. [1989]. "Precision architecture," *Computer* 22:1 (January), 78–91.

Levy, H., and R. Eckhouse [1989]. *Computer Programming and Architecture: The VAX*, Digital Press, Boston.

Lindholm, T., and F. Yellin [1999]. *The Java Virtual Machine Specification*, 2nd ed., Addison-Wesley, Reading, Mass.

Lunde, A. [1977]. "Empirical evaluation of some features of instruction set processor architecture," *Communications of the ACM* 20:3 (March), 143–152.

Magenheimer, D. J., L. Peters, K. W. Pettis, and D. Zuras [1988]. "Integer multiplication and division on the HP precision architecture," *IEEE Trans. on Computers* 37:8, 980–990.

McGhan, H., and M. O'Connor [1998]. "PicoJava: A direct execution engine for Java bytecode," *Computer* 31:10 (October), 22–30.

McKeeman, W. M. [1967]. "Language directed computer design," *Proc. AFIPS Fall Joint Computer Conf.*, November 14–16, 1967, Washington, D.C., 413–417.

Meyers, G. J. [1978]. "The evaluation of expressions in a storage-to-storage architecture," *Computer Architecture News* 7:3 (October), 20–23.

Meyers, G. J. [1982]. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.

MIPS. [1997]. *MIPS16 Application Specific Extension Product Description*.

Mitsubishi. [1996]. *Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual*, Mitsubishi, Cypress, Calif.

Morse, S., B. Ravenal, S. Mazor, and W. Pohlman [1980]. "Intel microprocessors—8080 to 8086," *Computer* 13:10 (October).

Moussouris, J., L. Crudele, D. Freitas, C. Hansen, E. Hudson, S. Przybylski, T. Riordan, and C. Rowen [1986]. "A CMOS RISC processor with integrated system functions," *Proc. IEEE COMPCON*, March 3–6, 1986, San Francisco, 191.

Muchnick, S. S. [1988]. "Optimizing compilers for SPARC," *Sun Technology* 1:3 (Summer), 64–77.

Palmer, J., and S. Morse [1984]. *The 8087 Primer*, John Wiley & Sons, New York, 93.

Patterson, D. [1985]. "Reduced instruction set computers," *Communications of the ACM* 28:1 (January), 8–21.

Patterson, D. A., and D. R. Ditzel [1980]. "The case for the reduced instruction set computer," *Computer Architecture News* 8:6 (October), 25–33.

Patterson, D. A., P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel, and K. Van Dyke [1983]. "Architecture of a VLSI instruction cache for a RISC," *10th Annual Int'l. Conf. on Computer Architecture Conf. Proc.*, June 13–16, 1983, Stockholm, Sweden, 108–116.

Radin, G. [1982]. "The 801 minicomputer," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 39–47.

Riemens, A., K. A. Vissers, R. J. Schutten, F. W. Sijstermans, G. J. Hekstra, and G. D. La Hei [1999]. "Trimedia CPU64 application domain and benchmark suite," *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, October 10–13, 1999, Austin, Tex., 580–585.

Ropers, A., H. W. Lollman, and J. Wellhausen [1999]. *DSPstone: Texas Instruments TMS320C54x*, Tech. Rep. Nr. IB 315 1999/9-ISS-Version 0.9, Aachen University of Technology, Aachen, Germany (*www.ert.rwth-aachen.de/Projekte/Tools/coal/dspstone_c54x/index.html*).

Shustek, L. J. [1978]. "Analysis and Performance of Computer Instruction Sets," Ph.D. dissertation, Stanford University, Palo Alto, Calif.

Silicon Graphics. [1996]. *MIPS V Instruction Set* (see *http://www.sgi.com/MIPS/arch/ISA5/#MIPSV_indx*).

Sites, R. L., and R. Witek, eds. [1995]. *Alpha Architecture Reference Manual*, 2nd ed., Digital Press, Newton, Mass.

Strauss, W. [1998]. "DSP Strategies 2002," *www.usadata.com/market_research/spr_05/spr_r127-005.htm*.

Strecker, W. D. [1978]. "VAX-11/780: A virtual address extension of the PDP-11 family," *Proc. AFIPS National Computer Conf.*, June 5–8, 1978, Anaheim, Calif., 47, 967–980.

Sun Microsystems. [1989]. *The SPARC Architectural Manual*, Version 8, Part No. 800-1399-09, Sun Microsystems, Santa Clara, Calif.

Tanenbaum, A. S. [1978]. "Implications of structured programming for machine architecture," *Communications of the ACM* 21:3 (March), 237–246.

Taylor, G., P. Hilfinger, J. Larus, D. Patterson, and B. Zorn [1986]. "Evaluation of the SPUR LISP architecture," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo.

Texas Instruments [2000]. "History of innovation: 1980s," *www.ti.com/corp/docs/company/history/1980s.shtml*.

Thornton, J. E. [1964]. "Parallel operation in Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf., Part II*, October 27–29, 1964, San Francisco, 26, 33–40.

Ungar, D., R. Blau, P. Foley, D. Samples, and D. Patterson [1984]. "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 188–197.

van Eijndhoven, J. T. J., F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken [1999]. "Trimedia CPU64 architecture," *Proc. IEEE Int'l. Conf. on Computer Design: VLSI in Computers and Processors (ICCD'99)*, October 10–13, 1999, Austin, Tex., 586–592.

Wakerly, J. [1989]. *Microcomputer Architecture and Programming*, Wiley, New York.

Waters, F. (ed.) [1986]. *IBM RT Personal Computer Technology*, SA 23-1057, IBM, Austin, Tex.

Weaver, D. L., and T. Germond [1994]. *The SPARC Architectural Manual*, Version 9,

Prentice Hall, Englewood Cliffs, N.J.

Weiss, S., and J. E. Smith [1994]. *Power and PowerPC*, Morgan Kaufmann, San Francisco.

Wiecek, C. [1982]. "A case study of the VAX 11 instruction set usage for compiler execution," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 177–184.

Wulf, W. [1981]. "Compilers and computer architecture," *Computer* 14:7 (July), 41–47.

M.5 The Development of Pipelining and Instruction-Level Parallelism (Chapter 3 and Appendices C and H)

Early Pipelined CPUs

The first general-purpose pipelined processor is considered to be Stretch, the IBM 7030. Stretch followed the IBM 704 and had a goal of being 100 times faster than the 704. The goal was a stretch from the state of the art at that time, hence the nickname. The plan was to obtain a factor of 1.6 from overlapping fetch, decode, and execute, using a four-stage pipeline. Bloch [1959] and Bucholtz [1962] described the design and engineering trade-offs, including the use of ALU bypasses.

A series of general pipelining descriptions that appeared in the late 1970s and early 1980s provided most of the terminology and described most of the basic techniques used in simple pipelines. These surveys include Keller [1975], Ramamoorthy and Li [1977], and Chen [1980], as well as Kogge [1981], whose book is devoted entirely to pipelining. Davidson and his colleagues [1971, 1975] developed the concept of pipeline reservation tables as a design methodology for multicycle pipelines with feedback (also described in Kogge [1981]). Many designers use a variation of these concepts, in either designing pipelines or in creating software to schedule them.

The RISC processors were originally designed with ease of implementation and pipelining in mind. Several of the early RISC papers, published in the early 1980s, attempt to quantify the performance advantages of the simplification in instruction set. The best analysis, however, is a comparison of a VAX and a MIPS implementation published by Bhandarkar and Clark in 1991, 10 years after the first published RISC papers (see Figure M.1). After 10 years of arguments about the implementation benefits of RISC, this paper convinced even the most skeptical designers of the advantages of a RISC instruction set architecture.

J. E. Smith and his colleagues have written a number of papers examining instruction issue, exception handling, and pipeline depth for high-speed scalar CPUs. Kunkel and Smith [1986] evaluated the impact of pipeline overhead and dependences on the choice of optimal pipeline depth; they also provided an excellent discussion of latch design and its impact on pipelining. Smith and Pleszkun [1988] evaluated a variety of techniques for preserving precise exceptions. Weiss and Smith [1984] evaluated a variety of hardware pipeline scheduling and instruction issue techniques.

The MIPS R4000 was one of the first deeply pipelined microprocessors and is described by Killian [1991] and by Heinrich [1993]. The initial Alpha implementation (the 21064) has a similar instruction set and similar integer pipeline structure, with more pipelining in the floating-point unit.

The Introduction of Dynamic Scheduling

In 1964, CDC delivered the first CDC 6600. The CDC 6600 was unique in many ways. In addition to introducing scoreboarding, the CDC 6600 was the first processor to make extensive use of multiple functional units. It also had peripheral processors that used multithreading. The interaction between pipelining and instruction set design was understood, and a simple, load-store instruction set was used to promote pipelining. The CDC 6600 also used an advanced packaging technology. Thornton [1964] described the pipeline and I/O processor architecture, including the concept of out-of-order instruction execution. Thornton's book [1970] provides an excellent description of the entire processor, from technology to architecture, and includes a foreword by Cray. (Unfortunately, this book is currently out of print.) The CDC 6600 also has an instruction scheduler for the FORTRAN compilers, described by Thorlin [1967].

The IBM 360 Model 91: A Landmark Computer

The IBM 360/91 introduced many new concepts, including tagging of data, register renaming, dynamic detection of memory hazards, and generalized forwarding. Tomasulo's algorithm is described in his 1967 paper. Anderson, Sparacio, and Tomasulo [1967] described other aspects of the processor, including the use of branch prediction. Many of the ideas in the 360/91 faded from use for nearly 25 years before being broadly resurrected in the 1990s. Unfortunately, the 360/91 was not successful, and only a handful were sold. The complexity of the design made it late to the market and allowed the Model 85, which was the first IBM processor with a cache, to outperform the 91.

Branch-Prediction Schemes

The 2-bit dynamic hardware branch-prediction scheme was described by J. E. Smith [1981]. Ditzel and McLellan [1987] described a novel branch-target buffer for CRISP, which implements branch folding. The correlating predictor we examine was described by Pan, So, and Rameh [1992]. Yeh and Patt [1992, 1993] generalized the correlation idea and described multilevel predictors that use branch histories for each branch, similar to the local history predictor used in the 21264. McFarling's tournament prediction scheme, which he refers to as a combined predictor, is described in his 1993 technical report. There are a variety of more recent papers on branch prediction based on variations in the multilevel and correlating predictor ideas. Kaeli and Emma [1991] described return address prediction, and Evers et al. [1998] provided an in-depth analysis of multilevel predictors. The data shown in Chapter 3 are from Skadron et al. [1999]. There are several schemes for prediction that may offer some additional benefit beyond tournament predictors. Eden and Mudge [1998] and Jimenez and Lin [2002] have described such approaches.

The Development of Multiple-Issue Processors

IBM did pioneering work on multiple issue. In the 1960s, a project called ACS was under way in California. It included multiple-issue concepts, a proposal for dynamic scheduling (although with a simpler mechanism than Tomasulo's scheme, which used backup registers), and fetching down both branch paths. The project originally started as a new architecture to follow Stretch and surpass the CDC 6600/6800. ACS started in New York but was moved to California, later changed to be S/360 compatible, and eventually canceled. John Cocke was one of the intellectual forces behind the team that included a number of IBM veterans and younger contributors, many of whom went on to other important roles in IBM and elsewhere: Jack Bertram, Ed Sussenguth, Gene Amdahl, Herb Schorr, Fran Allen, Lynn Conway, and Phil Dauber, among others. While the compiler team published many of their ideas and had great influence outside IBM, the architecture ideas were not widely disseminated at that time. The most complete accessible documentation of this important project is at *www.cs.clemson.edu/~mark/acs.html*, which includes interviews with the ACS veterans and pointers to other sources. Sussenguth [1999] is a good overview of ACS.

Most of the early multiple-issue processors that actually reached the market followed an LIW or VLIW design approach. Charlesworth [1981] reported on the Floating Point Systems AP-120B, one of the first wide-instruction processors containing multiple operations per instruction. Floating Point Systems applied the concept of software pipelining both in a compiler and by handwriting assembly language libraries to use the processor efficiently. Because the processor was an attached processor, many of the difficulties of implementing multiple issue in general-purpose processors (for example, virtual memory and exception handling) could be ignored.

One of the interesting approaches used in early VLIW processors, such as the AP-120B and i860, was the idea of a pipeline organization that requires operations to be "pushed through" a functional unit and the results to be caught at the end of the pipeline. In such processors, operations advance only when another operation pushes them from behind (in sequence). Furthermore, an instruction specifies the destination for an instruction issued earlier that will be pushed out of the pipeline when this new operation is pushed in. Such an approach has the advantage that it does not specify a result destination when an operation first issues but only when the result register is actually written. This separation eliminates the need to detect write after write (WAW) and write after read (WAR) hazards in the hardware. The disadvantage is that it increases code size since no-ops may be needed to push results out when there is a dependence on an operation that is still in the pipeline and no other operations of that type are immediately needed. Instead of the "push-and-catch" approach used in these two processors, almost all designers have chosen to use *self-draining pipelines* that specify the destination in the issuing instruction and in which an issued instruction will complete without further action. The advantages in code density and simplifications in code generation seem to outweigh the advantages of the more unusual structure.

Several research projects introduced some form of multiple issue in the mid-1980s. For example, the Stanford MIPS processor had the ability to place two operations in a single instruction, although this capability was dropped in commercial variants of the architecture, primarily for performance reasons. Along with his colleagues at Yale, Fisher [1983] proposed creating a processor with a very wide instruction (512 bits) and named this type of processor a VLIW. Code was generated for the processor using trace scheduling, which Fisher [1981] had developed originally for generating horizontal microcode. The implementation of trace scheduling for the Yale processor is described by Fisher et al. [1984] and by Ellis [1986].

Although IBM canceled ACS, active research in the area continued in the 1980s. More than 10 years after ACS was canceled, John Cocke made a new proposal for a superscalar processor that dynamically made issue decisions; he and Tilak Agerwala described the key ideas in several talks in the mid-1980s and coined the term *superscalar*. He called the design America; it is described by Agerwala and Cocke [1987]. The IBM Power1 architecture (the RS/6000 line) is based on these ideas (see Bakoglu et al. [1989]).

J. E. Smith [1984] and his colleagues at Wisconsin proposed the decoupled approach that included multiple issue with limited dynamic pipeline scheduling. A key feature of this processor is the use of queues to maintain order among a class of instructions (such as memory references) while allowing it to slip behind or ahead of another class of instructions. The Astronautics ZS-1 described by Smith et al. [1987] embodies this approach with queues to connect the load-store unit and the operation units. The Power2 design uses queues in a similar fashion. J. E. Smith [1989] also described the advantages of dynamic scheduling and compared that approach to static scheduling.

The concept of speculation has its roots in the original 360/91, which performed a very limited form of speculation. The approach used in recent processors combines the dynamic scheduling techniques of the 360/91 with a buffer to allow in-order commit. Smith and Pleszkun [1988] explored the use of buffering to maintain precise interrupts and described the concept of a reorder buffer. Sohi [1990] described adding renaming and dynamic scheduling, making it possible to use the mechanism for speculation. Patt and his colleagues were early proponents of aggressive reordering and speculation. They focused on checkpoint and restart mechanisms and pioneered an approach called HPSm, which is also an extension of Tomasulo's algorithm [Hwu and Patt 1986].

The use of speculation as a technique in multiple-issue processors was evaluated by Smith, Johnson, and Horowitz [1989] using the reorder buffer technique; their goal was to study available ILP in nonscientific code using speculation and multiple issue. In a subsequent book, Johnson [1990] described the design of a speculative superscalar processor. Johnson later led the AMD K-5 design, one of the first speculative superscalars.

In parallel with the superscalar developments, commercial interest in VLIW approaches also increased. The Multiflow processor (see Colwell et al. [1987]) was based on the concepts developed at Yale, although many important refinements were made to increase the practicality of the approach. Among these was a control-lable store buffer that provided support for a form of speculation. Although more than 100 Multiflow processors were sold, a variety of problems, including the difficulties of introducing a new instruction set from a small company and competition from commercial RISC microprocessors that changed the economics in the mini-computer market, led to the failure of Multiflow as a company.

Around the same time as Multiflow, Cydrome was founded to build a VLIW-style processor (see Rau et al. [1989]), which was also unsuccessful commercially. Dehnert, Hsu, and Bratt [1989] explained the architecture and performance of the

Cydrome Cydra 5, a processor with a wide-instruction word that provides dynamic register renaming and additional support for software pipelining. The Cydra 5 is a unique blend of hardware and software, including conditional instructions and register rotation, aimed at extracting ILP. Cydrome relied on more hardware than the Multiflow processor and achieved competitive performance primarily on vector-style codes. In the end, Cydrome suffered from problems similar to those of Multiflow and was not a commercial success. Both Multiflow and Cydrome, although unsuccessful as commercial entities, produced a number of people with extensive experience in exploiting ILP as well as advanced compiler technology; many of those people have gone on to incorporate their experience and the pieces of the technology in newer processors. Fisher and Rau [1993] edited a comprehensive collection of papers covering the hardware and software of these two important processors.

Rau had also developed a scheduling technique called *polycyclic scheduling*, which is a basis for most software-pipelining schemes (see Rau, Glaeser, and Picard [1982]). Rau's work built on earlier work by Davidson and his colleagues on the design of optimal hardware schedulers for pipelined processors. Other historical LIW processors have included the Apollo DN 10000 and the Intel i860, both of which could dual-issue FP and integer operations.

Compiler Technology and Hardware Support for Scheduling

Loop-level parallelism and dependence analysis were developed primarily by D. Kuck and his colleagues at the University of Illinois in the 1970s. They also coined the commonly used terminology of *antidependence* and *output dependence* and developed several standard dependence tests, including the GCD and Banerjee tests. The latter test was named after Uptal Banerjee and comes in a variety of flavors. Recent work on dependence analysis has focused on using a variety of exact tests ending with a linear programming algorithm called Fourier–Motzkin. D. Maydan and W. Pugh both showed that the sequences of exact tests were a practical solution.

In the area of uncovering and scheduling ILP, much of the early work was connected to the development of VLIW processors, described earlier. Lam [1988] developed algorithms for software pipelining and evaluated their use on Warp, a wide-instruction-word processor designed for special-purpose applications. Weiss and Smith [1987] compared software pipelining versus loop unrolling as techniques for scheduling code on a pipelined processor. Rau [1994] developed modulo scheduling to deal with the issues of software-pipelining loops and simultaneously handling register allocation.

Support for speculative code scheduling was explored in a variety of contexts, including several processors that provided a mode in which exceptions were ignored, allowing more aggressive scheduling of loads (e.g., the MIPS TFP processor [Hsu 1994]). Several groups explored ideas for more aggressive hardware support for speculative code scheduling. For example, Smith, Horowitz, and Lam

[1992] created a concept called boosting that contains a hardware facility for supporting speculation but provides a checking and recovery mechanism, similar to those in IA-64 and Crusoe. The sentinel scheduling idea, which is also similar to the speculate-and-check approach used in both Crusoe and the IA-64 architectures, was developed jointly by researchers at the University of Illinois and HP Laboratories (see Mahlke et al. [1992]).

In the early 1990s, Wen-Mei Hwu and his colleagues at the University of Illinois developed a compiler framework, called IMPACT (see Chang et al. [1991]), for exploring the interaction between multiple-issue architectures and compiler technology. This project led to several important ideas, including superblock scheduling (see Hwu et al. [1993]), extensive use of profiling for guiding a variety of optimizations (e.g., procedure inlining), and the use of a special buffer (similar to the ALAT or program-controlled store buffer) for compile-aided memory conflict detection (see Gallagher et al. [1994]). They also explored the performance trade-offs between partial and full support for predication in Mahlke et al. [1995].

The early RISC processors all had delayed branches, a scheme inspired from microprogramming, and several studies on compile time branch prediction were inspired by delayed branch mechanisms. McFarling and Hennessy [1986] did a quantitative comparison of a variety of compile time and runtime branch-prediction schemes. Fisher and Freudenberger [1992] evaluated a range of compile time branch-prediction schemes using the metric of distance between mispredictions. Ball and Larus [1993] and Calder et al. [1997] described static prediction schemes using collected program behavior.

EPIC and the IA-64 Development

The roots of the EPIC approach lie in earlier attempts to build LIW and VLIW machines—especially those at Cydrome and Multiflow—and in a long history of compiler work that continued after these companies failed at HP, the University of Illinois, and elsewhere. Insights gained from that work led designers at HP to propose a VLIW-style, 64-bit architecture to follow the HP PA RISC architecture. Intel was looking for a new architecture to replace the x86 (now called IA-32) architecture and to provide 64-bit capability. In 1995, they formed a partnership to design a new architecture, IA-64 (see Huck et al. [2000]), and build processors based on it. Itanium (see Sharangpani and Arora [2000]) is the first such processor. In 2002, Intel introduced the second-generation IA-64 design, the Itanium 2 (see McNairy and Soltis [2003] and McCormick and Knies [2002]).

Studies of ILP and Ideas to Increase ILP

A series of early papers, including Tjaden and Flynn [1970] and Riseman and Foster [1972], concluded that only small amounts of parallelism could be available at the instruction level without investing an enormous amount of hardware. These papers dampened the appeal of multiple instruction issue for more than 10 years.

Nicolau and Fisher [1984] published a paper based on their work with trace scheduling and asserted the presence of large amounts of potential ILP in scientific programs.

Since then there have been many studies of the available ILP. Such studies have been criticized because they presume some level of both hardware support and compiler technology. Nonetheless, the studies are useful to set expectations as well as to understand the sources of the limitations. Wall has participated in several such studies, including Jouppi and Wall [1989] and Wall [1991, 1993]. Although the early studies were criticized as being conservative (e.g., they didn't include speculation), the last study is by far the most ambitious study of ILP to date and the basis for the data in Section 3.10. Sohi and Vajapeyam [1989] provided measurements of available parallelism for wide-instruction-word processors. Smith, Johnson, and Horowitz [1989] also used a speculative superscalar processor to study ILP limits. At the time of their study, they anticipated that the processor they specified was an upper bound on reasonable designs. Recent and upcoming processors, however, are likely to be at least as ambitious as their processor. Skadron et al. [1999] examined the performance trade-offs and limitations in a processor comparable to the most aggressive processors in 2005, concluding that the larger window sizes will not make sense without significant improvements on branch prediction for integer programs.

Lam and Wilson [1992] looked at the limitations imposed by speculation and showed that additional gains are possible by allowing processors to speculate in multiple directions, which requires more than one PC. (Such schemes cannot exceed what perfect speculation accomplishes, but they help close the gap between realistic prediction schemes and perfect prediction.) Wall's 1993 study includes a limited evaluation of this approach (up to eight branches are explored).

Going Beyond the Data Flow Limit

One other approach that has been explored in the literature is the use of value prediction. Value prediction can allow speculation based on data values. There have been a number of studies of the use of value prediction. Lipasti and Shen published two papers in 1996 evaluating the concept of value prediction and its potential impact on ILP exploitation. Calder, Reinman, and Tullsen [1999] explored the idea of selective value prediction. Sodani and Sohi [1997] approached the same problem from the viewpoint of reusing the values produced by instructions. Moshovos et al. [1997] showed that deciding when to speculate on values, by tracking whether such speculation has been accurate in the past, is important to achieving performance gains with value speculation. Moshovos and Sohi [1997] and Chrysos and Emer [1998] focused on predicting memory dependences and using this information to eliminate the dependence through memory. González and González [1998], Babbay and Mendelson [1998], and Calder, Reinman, and Tullsen [1999] are more recent studies of the use of value prediction. This area is currently highly active, with new results being published in every conference.

Recent Advanced Microprocessors

The years 1994 and 1995 saw the announcement of wide superscalar processors (three or more issues per clock) by every major processor vendor: Intel Pentium Pro and Pentium II (these processors share the same core pipeline architecture, described by Colwell and Steck [1995]); AMD K-5, K-6, and Athlon; Sun UltraSPARC (see Lauterbach and Horel [1999]); Alpha 21164 (see Edmondson et al. [1995]) and 21264 (see Kessler [1999]); MIPS R10000 and R12000 (see Yeager [1996]); PowerPC 603, 604, and 620 (see Diep, Nelson, and Shen [1995]); and HP 8000 (Kumar [1997]). The latter part of the decade (1996–2000) saw second generations of many of these processors (Pentium III, AMD Athlon, and Alpha 21264, among others). The second generation, although similar in issue rate, could sustain a lower CPI and provided much higher clock rates. All included dynamic scheduling, and they almost universally supported speculation. In practice, many factors, including the implementation technology, the memory hierarchy, the skill of the designers, and the type of applications benchmarked, all play a role in determining which approach is best.

The period from 2000 to 2005 was dominated by three trends among superscalar processors: the introduction of higher clock rates achieved through deeper pipelining (e.g., in the Pentium 4; see Hinton et al. [2001]), the introduction of multithreading by IBM in the Power 4 and by Intel in the Pentium 4 Extreme, and the beginning of the movement to multicore by IBM in the Power 4, AMD in Opteron (see Keltcher et al. [2003]), and most recently by Intel (see Douglas [2005]).

Multithreading and Simultaneous Multithreading

The concept of multithreading dates back to one of the earliest transistorized computers, the TX-2. TX-2 is also famous for being the computer on which Ivan Sutherland created Sketchpad, the first computer graphics system. TX-2 was built at MIT's Lincoln Laboratory and became operational in 1959. It used multiple threads to support fast context switching to handle I/O functions. Clark [1957] described the basic architecture, and Forgie [1957] described the I/O architecture. Multithreading was also used in the CDC 6600, where a fine-grained multithreading scheme with interleaved scheduling among threads was used as the architecture of the I/O processors. The HEP processor, a pipelined multiprocessor designed by Denelcor and shipped in 1982, used fine-grained multithreading to hide the pipeline latency as well as to hide the latency to a large memory shared among all the processors. Because the HEP had no cache, this hiding of memory latency was critical. Burton Smith, one of the primary architects, described the HEP architecture in a 1978 paper, and Jordan [1983] published a performance evaluation. The TERA processor extends the multithreading ideas and is described by Alverson et al. in a 1992 paper. The Niagara multithreading approach is similar to those of the HEP and TERA systems, although Niagara employs caches reducing the need for thread-based latency hiding.

In the late 1980s and early 1990s, researchers explored the concept of coarse-grained multithreading (also called *block multithreading*) as a way to tolerate latency, especially in multiprocessor environments. The SPARCLE processor in the Alewife system used such a scheme, switching threads whenever a highlatency exceptional event, such as a long cache miss, occurred. Agarwal et al. described SPARCLE in a 1993 paper. The IBM Pulsar processor uses similar ideas.

By the early 1990s, several research groups had arrived at two key insights. First, they realized that fine-grained multithreading was needed to get the maximum performance benefit, since in a coarse-grained approach, the overhead of thread switching and thread start-up (e.g., filling the pipeline from the new thread) negated much of the performance advantage (see Laudon, Gupta, and Horowitz [1994]). Second, several groups realized that to effectively use large numbers of functional units would require both ILP and thread-level parallelism (TLP). These insights led to several architectures that used combinations of multithreading and multiple issue. Wolfe and Shen [1991] described an architecture called XIMD that statically interleaves threads scheduled for a VLIW processor. Hirata et al. [1992] described a proposed processor for media use that combines a static superscalar pipeline with support for multithreading; they reported speedups from combining both forms of parallelism. Keckler and Dally [1992] combined static scheduling of ILP and dynamic scheduling of threads for a processor with multiple functional units. The question of how to balance the allocation of functional units between ILP and TLP and how to schedule the two forms of parallelism remained open.

When it became clear in the mid-1990s that dynamically scheduled superscalars would be delivered shortly, several research groups proposed using the dynamic scheduling capability to mix instructions from several threads on the fly. Yamamoto et al. [1994] appear to have published the first such proposal, though the simulation results for their multithreaded superscalar architecture use simplistic assumptions. This work was quickly followed by Tullsen, Eggers, and Levy [1995], who provided the first realistic simulation assessment and coined the term *simultaneous multithreading*. Subsequent work by the same group together with industrial coauthors addressed many of the open questions about SMT. For example, Tullsen et al. [1996] addressed questions about the challenges of scheduling ILP versus TLP. Lo et al. [1997] provided an extensive discussion of the SMT concept and an evaluation of its performance potential, and Lo et. al. [1998] evaluated database performance on an SMT processor. Tuck and Tullsen [2003] reviewed the performance of SMT on the Pentium 4.

The IBM Power4 introduced multithreading (see Tendler et al. [2002]), while the Power5 used simultaneous multithreading. Mathis et al. [2005] explored the performance of SMT in the Power5, while Sinharoy et al. [2005] described the system architecture.

References

Agarwal, A., J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin [1993]. "Sparcle: An evolutionary processor design for large-scale multiprocessors," *IEEE Micro* 13 (June), 48–61.

Agerwala, T., and J. Cocke [1987]. *High Performance Reduced Instruction Set Processors*, Tech. Rep. RC12434, IBM Thomas Watson Research Center, Yorktown Heights, N.Y.

Alverson, G., R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith [1992]. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," *Proc. ACM/IEEE Conf. on Supercomputing*, November 16–20, 1992, Minneapolis, Minn., 188–197.

Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo [1967]. "The IBM 360 Model 91: Processor philosophy and instruction handling," *IBM J. Research and Development* 11:1 (January), 8–24.

Austin, T. M., and G. Sohi [1992]. "Dynamic dependency analysis of ordinary programs," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 342–351.

Babbay, F., and A. Mendelson [1998]. "Using value prediction to increase the power of speculative execution hardware," *ACM Trans. on Computer Systems* 16:3 (August), 234–270.

Bakoglu, H. B., G. F. Grohoski, L. E. Thatcher, J. A. Kaeli, C. R. Moore, D. P. Tattle, W. E. Male, W. R. Hardell, D. A. Hicks, M. Nguyen Phu, R. K. Montoye, W. T. Glover, and S. Dhawan [1989]. "IBM second-generation RISC processor organization," *Proc. IEEE Int'l. Conf. on Computer Design*, October, Rye Brook, N.Y., 138–142.

Ball, T., and J. Larus [1993]. "Branch prediction for free," *Proc. ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, June 23–25, 1993, Albuquerque, N.M., 300–313.

Bhandarkar, D., and D. W. Clark [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 310–319.

Bhandarkar, D., and J. Ding [1997]. "Performance characterization of the Pentium Pro processor," *Proc. Third Int'l. Symposium on High Performance Computer Architecture*, February 1–5, 1997, San Antonio, Tex., 288–297.

Bloch, E. [1959]. "The engineering design of the Stretch computer," *Proc. Eastern Joint Computer Conf.*, December 1–3, 1959, Boston, Mass., 48–59.

Bucholtz, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York.

Calder, B., D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn [1997]. "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.* 19:1, 188–222.

Calder, B., G. Reinman, and D. M. Tullsen [1999]. "Selective value prediction," *Proc. 26th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 2–4, 1999, Atlanta, Ga.

Chang, P. P., S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu [1991]. "IMPACT: An architectural framework for multiple-instruction-issue processors," *Proc. 18th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 27–30, 1991, Toronto, Canada, 266–275.

Charlesworth, A. E. [1981]. "An approach to scientific array processing: The architecture design of the AP-120B/FPS-164 family," *Computer* 14:9 (September), 18–27.

Chen, T. C. [1980]. "Overlap and parallel processing," in *Introduction to Computer Architecture*, H. Stone, ed., Science Research Associates, Chicago, 427–486.

Chrysos, G. Z., and J. S. Emer [1998]. "Memory dependence prediction using store sets," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 142–153.

Clark, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 173–177.

Clark, W. A. [1957]. "The Lincoln TX-2 computer development," *Proc. Western Joint Computer Conference*, February 26–28, 1957, Los Angeles, 143–145.

Colwell, R. P., and R. Steck [1995]. "A 0.6 μm BiCMOS processor with dynamic execution." *Proc. of IEEE Int'l. Symposium on Solid State Circuits (ISSCC)*, February 15–17, 1995, San Francisco, 176–177.

Colwell, R. P., R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman [1987]. "A VLIW architecture for a trace scheduling compiler," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 180–192.

Cvetanovic, Z., and R. E. Kessler [2000]. "Performance analysis of the Alpha 21264-based Compaq ES40 system," *27th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 10–14, 2000, Vancouver, Canada, 192–202.

Davidson, E. S. [1971]. "The design and control of pipelined function generators," *Proc. IEEE Conf. on Systems, Networks, and Computers*, January 19–21, 1971, Oaxtepec, Mexico, 19–21.

Davidson, E. S., A. T. Thomas, L. E. Shar, and J. H. Patel [1975]. "Effective control for pipelined processors," *Proc. IEEE COMPCON*, February 25–27, 1975, San Francisco, 181–184.

Dehnert, J. C., P. Y.-T. Hsu, and J. P. Bratt [1989]. "Overlapped loop support on the Cydra 5," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, Mass., 26–39.

Diep, T. A., C. Nelson, and J. P. Shen [1995]. "Performance evaluation of the PowerPC 620 microarchitecture," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy.

Ditzel, D. R., and H. R. McLellan [1987]. "Branch folding in the CRISP microprocessor: Reducing the branch delay to zero," *Proc. 14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1987, Pittsburgh, Penn., 2–7.

Douglas, J. [2005]. "Intel 8xx series and Paxville Xeon-MP Microprocessors," paper presented at Hot Chips 17, August 14–16, 2005, Stanford University, Palo Alto, Calif.

Eden, A., and T. Mudge [1998]. "The YAGS branch prediction scheme," *Proc. of the 31st Annual ACM/IEEE Int'l. Symposium on Microarchitecture*, November 30–December 2, 1998, Dallas, Tex., 69–80.

Edmondson, J. H., P. I. Rubinfield, R. Preston, and V. Rajagopalan [1995]. "Superscalar instruction execution in the 21164 Alpha microprocessor," *IEEE Micro* 15:2, 33–43.

Ellis, J. R. [1986]. *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass.

Emer, J. S., and D. W. Clark [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 301–310.

Evers, M., S. J. Patel, R. S. Chappell, and Y. N. Patt [1998]. "An analysis of correlation and predictability: What makes two-level branch predictors work," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 52–61.

Fisher, J. A. [1981]. "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. on Computers* 30:7 (July), 478–490.

Fisher, J. A. [1983]. "Very long instruction word architectures and ELI-512," *10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 140–150.

Fisher, J. A., and S. M. Freudenberger [1992]. "Predicting conditional branches from previous runs of a program," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 85–95.

Fisher, J. A., and B. R. Rau [1993]. *Journal of Supercomputing*, January (special issue).

Fisher, J. A., J. R. Ellis, J. C. Ruttenberg, and A. Nicolau [1984]. "Parallel processing: A smart compiler and a dumb processor," *Proc. SIGPLAN Conf. on Compiler Construction*, June 17–22, 1984, Montreal, Canada, 11–16.

Forgie, J. W. [1957]. "The Lincoln TX-2 input-output system," *Proc. Western Joint Computer Conference*, February 26–28, 1957, Los Angeles, 156–160.

Foster, C. C., and E. M. Riseman [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

Gallagher, D. M., W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.W. Hwu [1994]. "Dynamic memory disambiguation using the memory conflict buffer," *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, Santa Jose, Calif., 183–193.

González, J., and A. González [1998]. "Limits of instruction level parallelism with data speculation," *Proc. Vector and Parallel Processing (VECPAR) Conf.*, June 21–23, 1998, Porto, Portugal, 585–598.

Heinrich, J. [1993]. *MIPS R4000 User's Manual*, Prentice Hall, Englewood Cliffs, N.J.

Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel [2001]. "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, February.

Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa [1992]. "An elementary processor architecture with simultaneous instruction issuing from multiple threads," *Proc. 19th Annual Int'l. Symposium* *on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 136–145.

Hopkins, M. [2000]. “A critical look at IA-64: Massive resources, massive ILP, but can it deliver?” *Microprocessor Report*, February.

Hsu, P. [1994]. “Designing the TFP microprocessor,” *IEEE Micro* 18:2 (April), 2333.

Huck, J. et al. [2000]. “Introducing the IA-64 Architecture” *IEEE Micro*, 20:5 (September–October), 12–23.

Hwu, W.-M., and Y. Patt [1986]. “HPSm, a high performance restricted data flow architecture having minimum functionality,” *13th Annual Int’l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 297–307.

Hwu, W. W., S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery [1993]. “The superblock: An effective technique for VLIW and superscalar compilation,” *J. Supercomputing* 7:1, 2 (March), 229–248.

IBM. [1990]. “The IBM RISC System/6000 processor” (collection of papers), *IBM J. Research and Development* 34:1 (January).

Jimenez, D. A., and C. Lin [2002]. “Neural methods for dynamic branch prediction,” *ACM Trans. Computer Sys* 20:4 (November), 369–397.

Johnson, M. [1990]. *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, N.J.

Jordan, H. F. [1983]. “Performance measurements on HEP—a pipelined MIMD computer,” *Proc. 10th Annual Int’l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 207–212.

Jouppi, N. P., and D. W. Wall [1989]. “Available instruction-level parallelism for superscalar and superpipelined processors,” *Proc. Third Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 272–282.

Kaeli, D. R., and P. G. Emma [1991]. “Branch history table prediction of moving target branches due to subroutine returns,” *Proc. 18th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 27–30, 1991, Toronto, Canada, 34–42.

Keckler, S. W., and W. J. Dally [1992]. “Processor coupling: Integrating compile time and runtime scheduling for parallelism,” *Proc. 19th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 202–213.

Keller, R. M. [1975]. “Look-ahead processors,” *ACM Computing Surveys* 7:4 (December), 177–195.

Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway [2003]. “The AMD Opteron processor for multiprocessor servers,” *IEEE Micro* 23:2 (March–April), 66–76.

Kessler, R. [1999]. “The Alpha 21264 microprocessor,” *IEEE Micro* 19:2 (March/April) 24–36.

Killian, E. [1991]. “MIPS R4000 technical overview–64 bits/100 MHz or bust,” *Hot Chips III Symposium Record*, August 26–27, 1991, Stanford University, Palo Alto, Calif., 1.6–1.19.

Kogge, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

Kumar, A. [1997]. "The HP PA-8000 RISC CPU," *IEEE Micro* 17:2 (March/April).

Kunkel, S. R., and J. E. Smith [1986]. "Optimal pipelining in supercomputers," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 404–414.

Lam, M. [1988]. "Software pipelining: An effective scheduling technique for VLIW processors," *SIGPLAN Conf. on Programming Language Design and Implementation*, June 22–24, 1988, Atlanta, Ga., 318–328.

Lam, M. S., and R. P. Wilson [1992]. "Limits of control flow on parallelism," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 46–57.

Laudon, J., A. Gupta, and M. Horowitz [1994]. "Interleaving: A multithreading technique targeting multiprocessors and workstations," *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, San Jose, Calif., 308–318.

Lauterbach, G., and T. Horel [1999]. "UltraSPARC-III: Designing third generation 64-bit performance," *IEEE Micro* 19:3 (May/June).

Lipasti, M. H., and J. P. Shen [1996]. "Exceeding the dataflow limit via value prediction," *Proc. 29th Int'l. Symposium on Microarchitecture*, December 2–4, 1996, Paris, France.

Lipasti, M. H., C. B. Wilkerson, and J. P. Shen [1996]. "Value locality and load value prediction," *Proc. Seventh Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass., 138–147.

Lo, J., L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh [1998]. "An analysis of database workload performance on simultaneous multithreaded processors," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 39–50.

Lo, J., S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen [1997]. "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading," *ACM Trans. on Computer Systems* 15:2 (August), 322–354.

Mahlke, S. A., W. Y. Chen, W.-M. Hwu, B. R. Rau, and M. S. Schlansker [1992]. "Sentinel scheduling for VLIW and superscalar processors," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 238–247.

Mahlke, S. A., R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu [1995]. "A comparison of full and partial predicated execution support for ILP processors," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 138–149.

Mathis, H. M., A. E. Mercias, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel [2005]. "Characterization of the multithreading (SMT) efficiency in Power5," *IBM J. of Research and Development*, 49:4/5 (July/September), 555–564.

McCormick, J., and A. Knies [2002]. "A brief analysis of the SPEC CPU2000 benchmarks on the Intel Itanium 2 processor," paper presented at Hot Chips 14, August 18–20, 2002, Stanford University, Palo Alto, Calif.

McFarling, S. [1993]. *Combining Branch Predictors*, WRL Technical Note TN-36, Digital Western Research Laboratory, Palo Alto, Calif.

McFarling, S., and J. Hennessy [1986]. "Reducing the cost of branches," *Proc. 13th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1986, Tokyo, 396–403.

McNairy, C., and D. Soltis [2003]. "Itanium 2 processor microarchitecture," *IEEE Micro* 23:2 (March–April), 44–55.

Moshovos, A., and G. S. Sohi [1997]. "Streamlining inter-operation memory communication via data dependence prediction," *Proc. 30th Annual Int'l. Symposium on Microarchitecture*, December 1–3, Research Triangle Park, N.C., 235–245.

Moshovos, A., S. Breach, T. N. Vijaykumar, and G. S. Sohi [1997]. "Dynamic speculation and synchronization of data dependences," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Nicolau, A., and J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November), 968–976.

Pan, S.-T., K. So, and J. T. Rameh [1992]. "Improving the accuracy of dynamic branch prediction using branch correlation," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 76–84.

Postiff, M.A., D. A. Greene, G. S. Tyson, and T. N. Mudge [1999]. "The limits of instruction level parallelism in SPEC95 applications," *Computer Architecture News* 27:1 (March), 31–40.

Ramamoorthy, C. V., and H. F. Li [1977]. "Pipeline architecture," *ACM Computing Surveys* 9:1 (March), 61–102.

Rau, B. R. [1994]. "Iterative modulo scheduling: An algorithm for software pipelining loops," *Proc. 27th Annual Int'l. Symposium on Microarchitecture*, November 30–December 2, 1994, San Jose, Calif., 63–74.

Rau, B. R., C. D. Glaeser, and R. L. Picard [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *Proc. Ninth Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 26–29, 1982, Austin, Tex., 131–139.

Rau, B. R., D. W. L. Yen, W. Yen, and R. A. Towle [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *IEEE Computers* 22:1 (January), 12–34.

Riseman, E. M., and C. C. Foster [1972]. "Percolation of code to enhance paralleled dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

Rymarczyk, J. [1982]. "Coding guidelines for pipelined processors," *Proc. Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 1–3, 1982, Palo Alto, Calif., 12–19.

Sharangpani, H., and K. Arora [2000]. "Itanium Processor Microarchitecture," *IEEE Micro*, 20:5 (September–October), 24–43.

Sinharoy, B., R. N. Koala, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner [2005]. "POWER5 system microarchitecture," *IBM J. of Research and Development*, 49:4–5, 505–521.

Sites, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023, Dept. of Computer Science, University of California, San Diego.

Skadron, K., P. S. Ahuja, M. Martonosi, and D. W. Clark [1999]. "Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques," *IEEE Trans. on Computers*, 48:11 (November).

Smith, A., and J. Lee [1984]. "Branch prediction strategies and branch-target buffer design," *Computer* 17:1 (January), 6–22.

Smith, B. J. [1978]. "A pipelined, shared resource MIMD computer," *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, August, Bellaire, Mich., 6–8.

Smith, J. E. [1981]. "A study of branch prediction strategies," *Proc. Eighth Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 12–14, 1981, Minneapolis, Minn., 135–148.

Smith, J. E. [1984]. "Decoupled access/execute computer architectures," *ACM Trans. on Computer Systems* 2:4 (November), 289–308.

Smith, J. E. [1989]. "Dynamic instruction scheduling and the Astronautics ZS-1," *Computer* 22:7 (July), 21–35.

Smith, J. E., and A. R. Pleszkun [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May), 562–573. (This paper is based on an earlier paper that appeared in *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass.)

Smith, J. E., G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, C. M. Rozewski, D. L. Fowler, K. R. Scidmore, and J. P. Laudon [1987]. "The ZS-1 central processor," *Proc. Second Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 199–204.

Smith, M. D., M. Horowitz, and M. S. Lam [1992]. "Efficient superscalar performance through boosting," *Proc. Fifth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 12–15, 1992, Boston, 248–259.

Smith, M. D., M. Johnson, and M. A. Horowitz [1989]. "Limits on multiple instruction issue," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 290–302.

Sodani, A., and G. Sohi [1997]. "Dynamic instruction reuse," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Sohi, G. S. [1990]. "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," *IEEE Trans. on Computers* 39:3 (March), 349–359.

Sohi, G. S., and S. Vajapeyam [1989]. "Tradeoffs in instruction format design for horizontal architectures," *Proc. Third Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 3–6, 1989, Boston, 15–25.

Sussenguth, E. [1999]. "IBM's ACS-1 Machine," *IEEE Computer* 22:11 (November).

Tendler, J. M., J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy [2002]. "Power4 system microarchitecture," *IBM J. of Research and Development*, 46:1, 5–26.

Thorlin, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Proc. Spring Joint Computer Conf.*, April 18–20, 1967, Atlantic City, N.J., 27.

Thornton, J. E. [1964]. "Parallel operation in the Control Data 6600," *Proc. AFIPS Fall Joint Computer Conf., Part II*, October 27–29, 1964, San Francisco, 26, 33–40.

Thornton, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, Ill.

Tjaden, G. S., and M. J. Flynn [1970]. "Detection and parallel execution of independent instructions," *IEEE Trans. on Computers* C-19:10 (October), 889–895.

Tomasulo, R. M. [1967]. "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. Research and Development* 11:1 (January), 25–33.

Tuck, N., and D. Tullsen [2003]. "Initial observations of the simultaneous multithreading Pentium 4 processor," *Proc. 12th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'03*), September 27–October 1, New Orleans, La., 26–34.

Tullsen, D. M., S. J. Eggers, and H. M. Levy [1995]. "Simultaneous multithreading: Maximizing on-chip parallelism," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 392–403.

Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm [1996]. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. 23rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 22–24, 1996, Philadelphia, Penn., 191–202.

Wall, D. W. [1991]. "Limits of instruction-level parallelism," *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 248–259.

Wall, D. W. [1993]. *Limits of Instruction-Level Parallelism*, Research Rep. 93/6, Western Research Laboratory, Digital Equipment Corp., Palo Alto, Calif.

Weiss, S., and J. E. Smith [1984]. "Instruction issue logic for pipelined supercomputers," *Proc. 11th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1984, Ann Arbor, Mich., 110–118.

Weiss, S., and J. E. Smith [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Int'l. Conf. on Architectural Support* *for Programming Languages and Operating Systems (ASPLOS)*, October 5–8, 1987, Palo Alto, Calif., 105–109.

Wilson, R. P., and M. S. Lam [1995]. “Efficient context-sensitive pointer analysis for C programs,” *Proc. ACM SIGPLAN’95 Conf. on Programming Language Design and Implementation*, June 18–21, 1995, La Jolla, Calif., 1–12.

Wolfe, A., and J. P. Shen [1991]. “A variable instruction stream extension to the VLIW architecture,” *Proc. Fourth Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 2–14.

Yamamoto, W., M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky [1994]. “Performance estimation of multistreamed, superscalar processors,” *Proc. 27th Annual Hawaii Int’l. Conf. on System Sciences*, January 4–7, 1994, Maui, 195–204.

Yeager, K. [1996]. “The MIPS R10000 superscalar microprocessor,” *IEEE Micro* 16:2 (April), 28–40.

Yeh, T., and Y. N. Patt [1992]. “Alternative implementations of two-level adaptive branch prediction,” *Proc. 19th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 124–134.

Yeh, T., and Y. N. Patt [1993]. “A comparison of dynamic branch predictors that use two levels of branch history,” *Proc. 20th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 16–19, 1993, San Diego, Calif., 257–266.

M.6 The Development of SIMD Supercomputers, Vector Computers, Multimedia SIMD Instruction Extensions, and Graphical Processor Units (Chapter 4)

In this historical section, we start with perhaps the most infamous supercomputer, the Illiac IV, as a representative of the early SIMD (Single Instruction, Multiple Data) architectures and then move to perhaps the most famous supercomputer, the Cray-1, as a representative of vector architectures. The next step is Multimedia SIMD Extensions, which got its name in part due to an advertising campaign involving the “Bunny People,” a disco-dancing set of workers in cleansuits on a semiconductor fabrication line. We conclude with the history of GPUs, which is not quite as colorful.

SIMD Supercomputers

The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components. … Centralizing the [control unit] gives rise to the basic organization of [an] … array processor such as the Illiac IV.

Bouknight et al. [1972]

… with Iliac IV, programming the machine was very difficult and the architecture probably was not very well suited to some of the applications we were trying to run. The key idea was that I did not think we had a very good match in Iliac IV between applications and architecture.

David Kuck
Software designer for the Illiac IV and
early pioneer in parallel software
David Kuck
An oral history conducted in 1991 by Andrew Goldstein,
IEEE History Center, New Brunswick, N.J.

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. Rather than pipelining the data computation as in vector architectures, these machines had an array of functional units; hence, they might be considered array processors.

The earliest ideas on SIMD-style computers are from Unger [1958] and Slotnick, Borck, and McReynolds [1962]. Slotnick's Solomon design formed the basis of the Illiac IV, perhaps the most infamous of the supercomputer projects. Although successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. (In 2011 dollars, that was an increase from $54M to $152M.) Actual performance was at best 15 MFLOPS versus initial predictions of 1000 MFLOPS for the full system [Hord 1982]. Delivered to NASA Ames Research in 1972, the computer required three more years of engineering before it was usable. These events slowed investigation of SIMD, but Danny Hillis [1985] resuscitated this style in the Connection Machine, which had 65,536 1-bit processors.

The basic trade-off in SIMD multiprocessors is performance of a processor versus number of processors. SIMD supercomputers of the 1980s emphasized a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single-bit-wide processors, while the Illiac IV planned for 64 64-bit processors. Massively parallel SIMD multiprocessors relied on interconnection or communication networks to exchange data between processing elements.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model faded away as supercomputers for two main reasons. First, it is too inflexible. A number of important problems were not data parallel, and the architecture did not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often have worse cost-performance compared with that of the alternatives. Second, SIMD could not take advantage of the tremendous performance and cost advantages of SISD (Single Instruction, Single Data) microprocessor technology of the 1980s, which was doubling in performance every 18 months. Instead of leveraging this low-cost technology, designers of SIMD multiprocessors had to build custom processors for their multiprocessors.

Vector Computers

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors. … One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

Seymour Cray

Public lecture at Lawrence Livermore Laboratories
on the introduction of the Cray-1 (1976)

The first vector processors were the Control Data Corporation (CDC) STAR-100 (see Hintz and Tate [1972]) and the Texas Instruments ASC (see Watson [1972]), both announced in 1972. Both were memory-memory vector processors. They had relatively slow scalar units—the STAR used the same units for scalars and vectors—making the scalar pipeline extremely deep. Both processors had high start-up overhead and worked on vectors of several hundred to several thousand elements. The crossover between scalar and vector could be over 50 elements. It appears that not enough attention was paid to the role of Amdahl's law on these two processors.

Seymour Cray, who worked on the 6600 and the 7600 at CDC, founded Cray Research and introduced the Cray-1 in 1976 (see Russell [1978]). The Cray-1 used a vector-register architecture to lower start-up overhead significantly and to reduce memory bandwidth requirements. He also had efficient support for non-unit stride and invented chaining. Most importantly, the Cray-1 was the fastest scalar processor in the world at that time. This matching of good scalar and vector performance was probably the most significant factor in making the Cray-1 a success. Some customers bought the processor primarily for its outstanding scalar performance. Many subsequent vector processors are based on the architecture of this first commercially successful vector processor. Baskett and Keller [1977] provided a good evaluation of the Cray-1.

In 1981, CDC started shipping the CYBER 205 (see Lincoln [1982]). The 205 had the same basic architecture as the STAR but offered improved performance all around as well as expandability of the vector unit with up to four lanes, each with multiple functional units and a wide load-store pipe that provided multiple words per clock. The peak performance of the CYBER 205 greatly exceeded the performance of the Cray-1; however, on real programs, the performance difference was much smaller.

In 1983, Cray Research shipped the first Cray X-MP (see Chen [1983]). With an improved clock rate (9.5 ns versus 12.5 ns on the Cray-1), better chaining support (allowing vector operations with RAW dependencies to operate in parallel), and multiple memory pipelines, this processor maintained the Cray Research lead in supercomputers. The Cray-2, a completely new design configurable with up to four processors, was introduced later. A major feature of the Cray-2 was the use of DRAM, which made it possible to have very large memories at the time. The first Cray-2, with its 256M word (64-bit words) memory, contained more memory than the total of all the Cray machines shipped to that point! The Cray-2 had a much faster clock than the X-MP, but also much deeper pipelines; however, it lacked chaining, had enormous memory latency, and had only one memory pipe per processor. In general, the Cray-2 was only faster than the Cray X-MP on problems that required its very large main memory.

That same year, processor vendors from Japan entered the supercomputer marketplace. First were the Fujitsu VP100 and VP200 (see Miura and Uchida [1983]), and later came the Hitachi S810 and the NEC SX/2 (see Watanabe [1987]). These processors proved to be close to the Cray X-MP in performance. In general, these three processors had much higher peak performance than the Cray X-MP. However, because of large start-up overhead, their typical performance was often lower than that of the Cray X-MP. The Cray X-MP favored a multiple-processor approach, first offering a two-processor version and later a four-processor version. In contrast, the three Japanese processors had expandable vector capabilities.

In 1988, Cray Research introduced the Cray Y-MP—a bigger and faster version of the X-MP. The Y-MP allowed up to eight processors and lowered the cycle time to 6 ns. With a full complement of eight processors, the Y-MP was generally the fastest supercomputer, though the single-processor Japanese supercomputers could be faster than a one-processor Y-MP. In late 1989, Cray Research was split into two companies, both aimed at building high-end processors available in the early 1990s. Seymour Cray headed the spin-off, Cray Computer Corporation, until its demise in 1995. Their initial processor, the Cray-3, was to be implemented in gallium arsenide, but they were unable to develop a reliable and cost-effective implementation technology. Shortly before his tragic death in a car accident in 1996, Seymour Cray started yet another company to develop high-performance systems but this time using commodity components.

Cray Research focused on the C90, a new high-end processor with up to 16 processors and a clock rate of 240 MHz. This processor was delivered in 1991. In 1993, Cray Research introduced their first highly parallel processor, the T3D, employing up to 2048 Digital Alpha21064 microprocessors. In 1995, they announced the availability of both a new low-end vector machine, the J90, and a high-end machine, the T90. The T90 was much like the C90, but with a clock that was twice as fast (460 MHz), using three-dimensional packaging and optical clock distribution.

In 1995, Cray Research was acquired by Silicon Graphics. In 1998, it released the SV1 system, which grafted considerably faster CMOS processors onto the J90 memory system. It also added a data cache for vectors to each CPU to help meet the increased memory bandwidth demands. Silicon Graphics sold Cray Research to Tera Computer in 2000, and the joint company was renamed Cray Inc.

The Japanese supercomputer makers continued to evolve their designs. In 2001, the NEC SX/5 was generally held to be the fastest available vector supercomputer, with 16 lanes clocking at 312 MHz and with up to 16 processors sharing the same memory. The NEC SX/6, released in 2001, was the first commercial single-chip vector microprocessor, integrating an out-of-order quad-issue superscalar processor, scalar instruction and data caches, and an eight-lane vector unit on a single die [Kitagawa et al. 2003]. The Earth Simulator is constructed from 640 nodes connected with a full crossbar, where each node comprises eight SX-6 vector microprocessors sharing a local memory. The SX-8, released in 2004, reduces the number of lanes to four but increases the vector clock rate to 2 GHz. The scalar unit runs at a slower 1 GHz clock rate, a common pattern in vector machines where the lack of hazards simplifies the use of deeper pipelines in the vector unit.

In 2002, Cray Inc. released the X1 based on a completely new vector ISA. The X1 SSP processor chip integrates an out-of-order superscalar with scalar caches running at 400 MHz and a two-lane vector unit running at 800 MHz. When four SSP chips are ganged together to form an MSP, the resulting peak vector performance of 12.8 GFLOPS is competitive with the contemporary NEC SX machines. The X1E enhancement, delivered in 2004, raises the clock rates to 565 and 1130 MHz, respectively. Many of the ideas were borrowed from the Cray T3E design, which is a MIMD (Multiple Instruction, Multiple Data) computer that uses off-the-shelf microprocessors. X1 has a new instruction set with a larger number of registers and with memory distributed locally with the processor in shared address space. The out-of-order scalar unit and vector units are decoupled, so that the scalar unit can get ahead of the vector unit. Vectors become shorter when the data are blocked to utilize the MSP caches, which is not a good match to an eight-lane vector unit. To handle these shorter vectors, each processor with just two vector lanes can work on a different loop.

The Cray X2 was announced in 2007, and it may prove to be the last Cray vector architecture to be built, as it's difficult to justify the investment in new silicon given the size of the market. The processor has a 1.3 GHz clock rate and 8 vector lanes for a processor peak performance of 42 GFLOP/sec for single precision. It includes both L1 and L2 caches. Each node is a 4-way SMP with up to 128 GBytes of DRAM, and the maximum size is 8K nodes.

The NEC SX-9 has up to 16 processors per node, with each processor having 8 lanes and running at 3.2 GHz. It was announced in 2008. The peak double precision vector performance is 102 GFLOP/sec. The 16 processor SMP can have 1024 GBytes of DRAM. The maximum size is 512 nodes.

The basis for modern vectorizing compiler technology and the notion of data dependence was developed by Kuck and his colleagues [1974] at the University of Illinois. Padua and Wolfe [1986] gave a good overview of vectorizing compiler technology.

Multimedia SIMD Instruction Extensions

What could a computer hardware company … possibly have in common with disco dancing. A lot, if one goes by an advertisement campaign released by the world's largest microprocessor company … Intel, in 1997.

IBS Center for Management Research

"Dancing Its Way Towards Leadership," 2002

Going through the history books, the 1957 TX-2 had partitioned ALUs to support media of the time, but these ideas faded away to be rediscovered 30 years later in the personal computer era. Since every desktop microprocessor by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations. Many graphics systems use 8 bits to represent each of the 3 primary colors plus 8 bits for a transparency of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and half words take up less space when stored in memory, but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there is little support beyond data transfers. The Intel i860 was justified as a graphical accelerator within the company. Its architects recognized that many graphics and audio applications would perform the same operation on vectors of these data [Atkins 1991; Kohn 1989]. Although a vector unit was beyond the transistor budget of the i860 in 1989, by partitioning the carry chains within a 64-bit ALU, it could perform simultaneous operations on short vectors of eight 8-bit operands, four 16-bit operands, or two 32-bit operands. The cost of such partitioned ALUs was small. Applications that lend themselves to such support include MPEG (video), video games (3D graphics), digital photography, and teleconferencing (audio and image processing).

Like a virus, over time such multimedia support has spread to nearly every desktop microprocessor. HP was the first successful desktop RISC to include such support, but soon every other manufacturer had their own take on the idea in the 1990s.

These extensions were originally called *subword parallelism* or *vector*. Since Intel marketing used SIMD to describe the MMX extension of the 80x86 announced in 1996, that became the popular name, due in part to a successful television advertising campaign involving disco dancers wearing clothing modeled after the cleansuits worn in semiconductor fabrication lines.

Graphical Processor Units

It's been almost three years since GPU computing broke into the mainstream of HPC with the introduction of NVIDIA's CUDA API in September 2007. Adoption of the technology since then has proceeded at a surprisingly strong and steady pace. Many organizations that began with small pilot projects a year or two ago have moved on to enterprise deployment, and GPU accelerated machines are now represented on the TOP500 list starting at position two. The relatively rapid adoption of CUDA by a community not known for the rapid adoption of much of anything is a noteworthy signal. Contrary to the accepted wisdom that GPU computing is more difficult, I believe its success thus far signals that it is no more complicated than good CPU programming. Further, it more clearly and succinctly expresses the parallelism of a large class of problems leading to code *that is easier to maintain, more scalable and better positioned to map to future many-core architectures.*

Vincent Natol

"Kudos for CUDA," HPCwire (2010)

3D graphics pipeline hardware evolved from the large expensive systems of the early 1980s to small workstations and then to PC accelerators in the mid- to late 1990s. During this period, three major transitions occurred:

- Performance-leading graphics subsystems declined in price from $50,000 to $200.
- Performance increased from 50 million pixels per second to 1 billion pixels per second and from 100,000 vertices per second to 10 million vertices per second.
- Native hardware capabilities evolved from wireframe (polygon outlines) to flat-shaded (constant color) filled polygons, to smooth-shaded (interpolated color) filled polygons, to full-scene anti-aliasing with texture mapping and rudimentary multitexturing.

Scalable GPUs

Scalability has been an attractive feature of graphics systems from the beginning. Workstation graphics systems gave customers a choice in pixel horse-power by varying the number of pixel processor circuit boards installed. Prior to the mid-1990s PC graphics scaling was almost nonexistent. There was one option—the VGA controller. As 3D-capable accelerators appeared, the market had room for a range of offerings. 3dfx introduced multiboard scaling with the original SLI (Scan Line Interleave) on their Voodoo2, which held the performance crown for its time (1998). Also in 1998, NVIDIA introduced distinct products as variants on a single architecture with Riva TNT Ultra (high-performance) and Vanta (low-cost), first by speed binning and packaging, then with separate chip designs (GeForce 2 GTS and GeForce 2 MX). At present, for a given architecture generation, four or five separate GPU chip designs are needed to cover the range of desktop PC performance and price points. In addition, there are separate segments in notebook and workstation systems. After acquiring 3dfx, NVIDIA continued the multi-GPU SLI concept in 2004, starting with GeForce 6800—providing multi-GPU scalability transparently to the programmer and to the user. Functional behavior is identical across the scaling range; one application will run unchanged on any implementation of an architectural family.

Graphics Pipelines

Early graphics hardware was configurable, but not programmable by the application developer. With each generation, incremental improvements were offered; however, developers were growing more sophisticated and asking for more new features than could be reasonably offered as built-in fixed functions. The NVIDIA

GeForce 3, described by Lindholm et al. [2001], took the first step toward true general shader programmability. It exposed to the application developer what had been the private internal instruction set of the floating-point vertex engine. This coincided with the release of Microsoft's DirectX 8 and OpenGL's vertex shader extensions. Later GPUs, at the time of DirectX 9, extended general programmability and floating-point capability to the pixel fragment stage and made texture available at the vertex stage. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel fragment processor programmed with DirectX 9 and OpenGL. The GeForce FX added 32-bit floating-point pixel processors. This was part of a general trend toward unifying the functionality of the different stages, at least as far as the application programmer was concerned. NVIDIA's GeForce 6800 and 7800 series were built with separate processor designs and separate hardware dedicated to the vertex and to the fragment processing. The XBox 360 introduced an early unified processor GPU in 2005, allowing vertex and pixel shaders to execute on the same processor.

GPGPU: An Intermediate Step

As DirectX 9-capable GPUs became available, some researchers took notice of the raw performance growth path of GPUs and began to explore the use of GPUs to solve complex parallel problems. DirectX 9 GPUs had been designed only to match the features required by the graphics API. To access the computational resources, a programmer had to cast their problem into native graphics operations. For example, to run many simultaneous instances of a pixel shader, a triangle had to be issued to the GPU (with clipping to a rectangle shape if that was what was desired). Shaders did not have the means to perform arbitrary scatter operations to memory. The only way to write a result to memory was to emit it as a pixel color value and configure the framebuffer operation stage to write (or blend, if desired) the result to a two-dimensional framebuffer. Furthermore, the only way to get a result from one pass of computation to the next was to write all parallel results to a pixel framebuffer, then use that framebuffer as a texture map as input to the pixel fragment shader of the next stage of the computation. Mapping general computations to a GPU in this era was quite awkward. Nevertheless, intrepid researchers demonstrated a handful of useful applications with painstaking efforts. This field was called "GPGPU" for general-purpose computing on GPUs.

GPU Computing

While developing the Tesla architecture for the GeForce 8800, NVIDIA realized its potential usefulness would be much greater if programmers could think of the GPU as a processor. NVIDIA selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload.

For the DirectX 10 generation, NVIDIA had already begun work on a highefficiency floating-point and integer processor that could run a variety of simultaneous workloads to support the logical graphics pipeline. This processor was designed to take advantage of the common case of groups of threads executing the same code path. NVIDIA added memory load and store instructions with integer byte addressing to support the requirements of compiled C programs. It introduced the thread block (cooperative thread array), grid of thread blocks, and barrier synchronization to dispatch and manage highly parallel computing work. Atomic memory operations were added. NVIDIA developed the CUDA C/C++ compiler, libraries, and runtime software to enable programmers to readily access the new data-parallel computation model and develop applications.

To create a vendor-neutral GPU programming language, a large number of companies are creating compilers for the OpenCL language, which has many of the features of CUDA but which runs on many more platforms. In 2011, the performance is much higher if you write CUDA code for GPUs than if you write OpenCL code.

AMD's acquisition of ATI, the second leading GPU vendor, suggests a spread of GPU computing. The AMD Fusion architecture, announced just as this edition was being finished, is an initial merger between traditional GPUs and traditional CPUs. NVIDIA also announced Project Denver, which combines an ARM scalar processor with NVIDIA GPUs in a single address space. When these systems are shipped, it will be interesting to learn just how tightly integrated they are and the impact of integration on performance and energy of both data parallel and graphics applications.

References

SIMD Supercomputers

Bouknight, W. J., S. A. Deneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick [1972]. "The Illiac IV system," *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 306–316.

Hillis, W. D. [1985]. *The Connection Multiprocessor*, MIT Press, Cambridge, Mass.

Hord, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds [1962]. "The Solomon computer," *Proc. AFIPS Fall Joint Computer Conf.*, December 4–6, 1962, Philadelphia, Penn., 97–107.

Unger, S. H. [1958]. "A computer oriented towards spatial problems," *Proc. Institute of Radio Engineers* 46:10 (October), 1744–1750.

Vector Architecture

Asanovic, K. [1998]. "Vector Microprocessors," Ph.D. thesis, Computer Science Division, University of California, Berkeley.

Baskett, F., and T. W. Keller [1977]. "An Evaluation of the Cray-1 Processor," in *High Speed Computer and Algorithm Organization*, D. J. Kuck, D. H. Lawrie, and A. H. Sameh, eds., Academic Press, San Diego, Calif., 71–84.

Chen, S. [1983]. “Large-scale and high-speed multiprocessor system for scientific applications,” *Proc. NATO Advanced Research Workshop on High Speed Computing*, June 20–22, Julich, West Germany. Also in K. Hwang, ed., “Superprocessors: Design and applications,” *IEEE*, August, 59–73, 1984.

Flynn, M. J. [1966]. “Very high-speed computing systems,” *Proc. IEEE* 54:12 (December), 1901–1909.

Gebis, J. and Patterson, D. [2007]. “Embracing and extending 20th-century instruction set architectures,” *IEEE Computer*, 40:4 (April), 68–75.

Hintz, R. G., and D. P. Tate [1972]. “Control data STAR-100 processor design,” *Proc. IEEE COMPCON*, September 12–14, 1972, San Francisco, 1–4.

Kitagawa, K., S. Tagaya, Y. Hagihara, and Y. Kanoh [2003]. “A hardware overview of SX-6 and SX-7 supercomputer,” *NEC Research and Development Journal* 44:1 (January), 2–7.

Kozyrakis, C., and D. Patterson [2002]. “Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks,” *Proc. 35th Annual Intl. Symposium on Microarchitecture (MICRO)*, November 18–22, 2002, Istanbul, Turkey.

Kuck, D., P. P. Budnik, S.-C. Chen, D. H. Lawrie, R. A. Towle, R. E. Strebendt, E. W. Davis, Jr., J. Han, P. W. Kraska, and Y. Muraoka [1974]. “Measurements of parallelism in ordinary Fortran programs,” *Computer* 7:1 (January), 37–46.

Lincoln, N. R. [1982]. “Technology and design trade offs in the creation of a modern supercomputer,” *IEEE Trans. on Computers* C-31:5 (May), 363–376.

Miura, K., and K. Uchida [1983]. “FACOM vector processing system: VP100/200,” *Proc. NATO Advanced Research Workshop on High Speed Computing*, June 20–22, Julich, West Germany. Also in K. Hwang, ed., “Superprocessors: Design and applications,” *IEEE*, August, 59–73, 1984.

Padua, D., and M. Wolfe [1986]. “Advanced compiler optimizations for supercomputers,” *Communications of the ACM* 29:12 (December), 1184–1201.

Russell, R. M. [1978]. “The Cray-1 processor system,” *Communications of the ACM* 21:1 (January), 63–72.

Vajapeyam, S. [1991]. “Instruction-Level Characterization of the Cray Y-MP Processor,” Ph.D. thesis, Computer Sciences Department, University of Wisconsin–Madison.

Watanabe, T. [1987]. “Architecture and performance of the NEC supercomputer SX system,” *Parallel Computing* 5, 247–255.

Watson, W. J. [1972]. “The TI ASC—a highly modular and flexible super processor architecture,” *Proc. AFIPS Fall Joint Computer Conf.*, December 5–7, 1972, Anaheim, Calif., 221–228.

Multimedia SIMD

Atkins, M. [1991]. “Performance and the i860 Microprocessor,” *IEEE Micro*, 11:5 (September), 24–27, 72–78.

Kohn, L., and N. Margulis [1989]. “Introducing the Intel i860 64-Bit Microprocessor,” *IEEE Micro*, 9:4 (July), 15–30.

GPU

Akeley, K., and T. Jermoluk [1988]. "High-performance polygon rendering," *Proc. SIGGRAPH 88*, August 1–5, 1988, Atlanta, Ga., 239–46.

Hillis, W. D., and G. L. Steele [1986]. "Data parallel algorithms," *Communications of the ACM* 29:12 (December), 1170–1183 (*http://doi.acm.org/10.1145/7902.7903*).

IEEE 754-2008 Working Group. [2006]. *DRAFT Standard for Floating-Point Arithmetic*, 754-2008 (https://doi.org/10.1109/IEEESTD.2008.4610935).

Lee, W. V., et al. [2010]. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *Proc. ISCA '10*, June 19–23, 2010, Saint-Malo, France.

Lindholm, E., M. J. Kligard, and H. Moreton [2001]. A user-programmable vertex engine. In *SIGGRAPH '01*: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 149–158.

Moore, G. E. [1965]. "Cramming more components onto integrated circuits," *Electronics* 38:8 (April 19), 114–117.

Williams, S., A. Waterman, and D. Patterson [2009]. "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, 52:4 (April), 65–76.

M.7 The History of Multiprocessors and Parallel Processing (Chapter 5 and Appendices F, G, and I)

There is a tremendous amount of history in multiprocessors; in this section, we divide our discussion by both time period and architecture. We start with the SIMD approach and the Illiac IV. We then turn to a short discussion of some other early experimental multiprocessors and progress to a discussion of some of the great debates in parallel processing. Next we discuss the historical roots of the present multiprocessors and conclude by discussing recent advances.

SIMD Computers: Attractive Idea, Many Attempts, No Lasting Successes

The cost of a general multiprocessor is, however, very high and further design options were considered which would decrease the cost without seriously degrading the power or efficiency of the system. The options consist of recentralizing one of the three major components. … Centralizing the [control unit] gives rise to the basic organization of [an] … array processor such as the Illiac IV.

Bouknight et al. [1972]

The SIMD model was one of the earliest models of parallel computing, dating back to the first large-scale multiprocessor, the Illiac IV. The key idea in that multiprocessor, as in more recent SIMD multiprocessors, is to have a single instruction that operates on many data items at once, using many functional units.

The earliest ideas on SIMD-style computers are from Unger [1958] and Slotnick, Borck, and McReynolds [1962]. Slotnick's Solomon design formed the basis of the Illiac IV, perhaps the most infamous of the supercomputer projects. Although successful in pushing several technologies that proved useful in later projects, it failed as a computer. Costs escalated from the $8 million estimate in 1966 to $31 million by 1972, despite construction of only a quarter of the planned multiprocessor. Actual performance was at best 15 MFLOPS versus initial predictions of 1000 MFLOPS for the full system [Hord 1982]. Delivered to NASA Ames Research in 1972, the computer took three more years of engineering before it was usable. These events slowed investigation of SIMD, but Danny Hillis [1985] resuscitated this style in the Connection Machine, which had 65,636 1-bit processors.

Real SIMD computers need to have a mixture of SISD and SIMD instructions. There is an SISD host computer to perform operations such as branches and address calculations that do not need parallel operation. The SIMD instructions are broadcast to all the execution units, each of which has its own set of registers. For flexibility, individual execution units can be disabled during an SIMD instruction. In addition, massively parallel SIMD multiprocessors rely on interconnection or communication networks to exchange data between processing elements.

SIMD works best in dealing with arrays in for loops; hence, to have the opportunity for massive parallelism in SIMD there must be massive amounts of data, or *data parallelism*. SIMD is at its weakest in case statements, where each execution unit must perform a different operation on its data, depending on what data it has. The execution units with the wrong data are disabled so that the proper units can continue. Such situations essentially run at 1/*n*th performance, where *n* is the number of cases.

The basic trade-off in SIMD multiprocessors is performance of a processor versus number of processors. Recent multiprocessors emphasize a large degree of parallelism over performance of the individual processors. The Connection Multiprocessor 2, for example, offered 65,536 single-bit-wide processors, while the Illiac IV had 64 64-bit processors.

After being resurrected in the 1980s, first by Thinking Machines and then by MasPar, the SIMD model has once again been put to bed as a general-purpose multiprocessor architecture, for two main reasons. First, it is too inflexible. A number of important problems cannot use such a style of multiprocessor, and the architecture does not scale down in a competitive fashion; that is, small-scale SIMD multiprocessors often have worse cost-performance compared with that of the alternatives. Second, SIMD cannot take advantage of the tremendous performance and cost advantages of microprocessor technology. Instead of leveraging this low-cost technology, designers of SIMD multiprocessors must build custom processors for their multiprocessors.

Although SIMD computers have departed from the scene as general-purpose alternatives, this style of architecture will continue to have a role in special-purpose designs. Many special-purpose tasks are highly data parallel and require a limited set of functional units. Thus, designers can build in support for certain operations, as well as hardwired interconnection paths among functional units. Such organizations are often called *array processors*, and they are useful for such tasks as image and signal processing.

Other Early Experiments

It is difficult to distinguish the first MIMD multiprocessor. Surprisingly, the first computer from the Eckert-Mauchly Corporation, for example, had duplicate units to improve availability. Holland [1959] gave early arguments for multiple processors. Two of the best-documented multiprocessor projects were undertaken in the 1970s at Carnegie Mellon University. The first of these was C.mmp [Wulf and Bell 1972; Wulf and Harbison 1978], which consisted of 16 PDP-11s connected by a crossbar switch to 16 memory units. It was among the first multiprocessors with more than a few processors, and it had a shared-memory programming model. Much of the focus of the research in the C.mmp project was on software, especially in the OS area. A later multiprocessor, Cm* [Swan et al. 1977], was a cluster-based multiprocessor with a distributed memory and a nonuniform access time. The absence of caches and a long remote access latency made data placement critical. This multiprocessor and a number of application experiments are well described by Gehringer, Siewiorek, and Segall [1987]. Many of the ideas in these multiprocessors would be reused in the 1980s when the microprocessor made it much cheaper to build multiprocessors.

Great Debates in Parallel Processing

The turning away from the conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer. … Electronic circuits are ultimately limited in their speed of operation by the speed of light … and many of the circuits were already operating in the nanosecond range.

Bouknight et al. [1972]

… sequential computers are approaching a fundamental physical limit on their potential computational power. Such a limit is the speed of light …

Angel L. DeCegama
The Technology of Parallel Processing, Vol. I (1989)

… today's multiprocessors … are nearing an impasse as technologies approach the speed of light. Even if the components of a sequential processor could be made to work this fast, the best that could be expected is no more than a few million instructions per second.

David Mitchell
The Transputer: The Time Is Now (1989)

The quotes above give the classic arguments for abandoning the current form of computing, and Amdahl [1967] gave the classic reply in support of continued focus on the IBM 360 architecture. Arguments for the advantages of parallel execution can be traced back to the 19th century [Menabrea 1842]! Yet, the effectiveness of the multiprocessor for reducing latency of individual important programs is still being explored. Aside from these debates about the advantages and limitations of parallelism, several hot debates have focused on how to build multiprocessors.

It's hard to predict the future, yet in 1989 Gordon Bell made two predictions for 1995. We included these predictions in the first edition of the book, when the outcome was completely unclear. We discuss them in this section, together with an assessment of the accuracy of the prediction.

The first was that a computer capable of sustaining a teraFLOPS—one million MFLOPS—would be constructed by 1995, using either a multicomputer with 4K to 32K nodes or a Connection Multiprocessor with several million processing elements [Bell 1989]. To put this prediction in perspective, each year the Gordon Bell Prize acknowledges advances in parallelism, including the fastest real program (highest MFLOPS). In 1989, the winner used an eight-processor Cray Y-MP to run at 1680 MFLOPS. On the basis of these numbers, multiprocessors and programs would have to have improved by a factor of 3.6 each year for the fastest program to achieve 1 TFLOPS in 1995. In 1999, the first Gordon Bell prize winner crossed the 1 TFLOPS bar. Using a 5832-processor IBM RS/6000 SST system designed specially for Livermore Laboratories, they achieved 1.18 TFLOPS on a shock-wave simulation. This ratio represents a year-to-year improvement of 1.93, which is still quite impressive.

What has become recognized since the 1990s is that, although we may have the technology to build a TFLOPS multiprocessor, it is not clear that the machine is cost effective, except perhaps for a few very specialized and critically important applications related to national security. We estimated in 1990 that to achieve 1 TFLOPS would require a machine with about 5000 processors and would cost about $100 million. The 5832-processor IBM system at Livermore cost $110 million. As might be expected, improvements in the performance of individual microprocessors both in cost and performance directly affect the cost and performance of large-scale multiprocessors, but a 5000-processor system will cost more than 5000 times the price of a desktop system using the same processor. Since that time, much faster multiprocessors have been built, but the major improvements have increasingly come from the processors in the past five years, rather than fundamental breakthroughs in parallel architecture.

The second Bell prediction concerned the number of data streams in supercomputers shipped in 1995. Danny Hillis believed that, although supercomputers with a small number of data streams may be the best sellers, the biggest multiprocessors would be multiprocessors with many data streams, and these would perform the bulk of the computations. Bell bet Hillis that in the last quarter of calendar year 1995 more sustained MFLOPS would be shipped in multiprocessors using few data streams ($\leq$100) rather than many data streams ($\geq$1000). This bet concerned only supercomputers, defined as multiprocessors costing more than $1 million and used for scientific applications. Sustained MFLOPS was defined for this bet as the number of floating-point operations per *month*, so availability of multiprocessors affects their rating.

In 1989, when this bet was made, it was totally unclear who would win. In 1995, a survey of the current publicly known supercomputers showed only six multiprocessors in existence in the world with more than 1000 data streams, so Bell's prediction was a clear winner. In fact, in 1995, much smaller microprocessor-based multiprocessors ($\leq$20 processors) were becoming dominant. In 1995, a survey of the 500 highest-performance multiprocessors in use (based on Linpack ratings), called the TOP500, showed that the largest number of multiprocessors were bus-based shared-memory multiprocessors! By 2005, various clusters or multicomputers played a large role. For example, in the top 25 systems, 11 were custom clusters, such as the IBM Blue Gene system or the Cray XT3; 10 were clusters of shared-memory multiprocessors (both using distributed and centralized memory); and the remaining 4 were clusters built using PCs with an off-the-shelf interconnect.

More Recent Advances and Developments

With the primary exception of the parallel vector multiprocessors (see Appendix G) and more recently of the IBM Blue Gene design, all other recent MIMD computers have been built from off-the-shelf microprocessors using a bus and logically central memory or an interconnection network and a distributed memory. A number of experimental multiprocessors built in the 1980s further refined and enhanced the concepts that form the basis for many of today's multiprocessors.

The Development of Bus-Based Coherent Multiprocessors

Although very large mainframes were built with multiple processors in the 1960s and 1970s, multiprocessors did not become highly successful until the 1980s. Bell [1985] suggested that the key was that the smaller size of the microprocessor allowed the memory bus to replace the interconnection network hardware and that portable operating systems meant that multiprocessor projects no longer required the invention of a new operating system. In his paper, Bell defined the terms *multiprocessor* and *multicomputer* and set the stage for two different approaches to building larger scale multiprocessors.

The first bus-based multiprocessor with snooping caches was the Synapse N + 1 described by Frank [1984]. Goodman [1983] wrote one of the first papers to describe snooping caches. The late 1980s saw the introduction of many commercial bus-based, snooping cache architectures, including the Silicon Graphics 4D/240 [Baskett, Jermoluk, and Solomon 1988], the Encore Multimax [Wilson 1987], and the Sequent Symmetry [Lovett and Thakkar 1988]. The mid-1980s

Name	Protocol type	Memory write policy	Unique feature	Multiprocessors using
Write Once	Write invalidate	Write-back after first write	First snooping protocol described in literature	
Synapse N + 1	Write invalidate	Write-back	Explicit state where memory is the owner	Synapse multiprocessors; first cache-coherent multiprocessors available
Berkeley (MOESI)	Write invalidate	Write-back	Owned shared state	Berkeley SPUR multiprocessor; Sun Enterprise servers
Illinois (MESI)	Write invalidate	Write-back	Clean private state; can supply data from any cache with a clean copy	SGI Power and Challenge series
“Firefly”	Write broadcast	Write-back when private, write through when shared	Memory updated on broadcast	No current multiprocessors; SPARCCenter 2000 closest

Figure M.2 Five snooping protocols summarized. Archibald and Baer [1986] use these names to describe the five protocols, and Eggers [1989] summarizes the similarities and differences as shown in this figure. The Firefly protocol was named for the experimental DEC Firefly multiprocessor, in which it appeared. The alternative names for protocols are based on the states they support: M = Modified, E = Exclusive (private clean), S = Shared, I = Invalid, O = Owner (shared dirty).

saw an explosion in the development of alternative coherence protocols, and Archibald and Baer [1986] provided a good survey and analysis, as well as references to the original papers. Figure M.2 summarizes several snooping cache coherence protocols and shows some multiprocessors that have used or are using that protocol.

The early 1990s saw the beginning of an expansion of such systems with the use of very wide, high-speed buses (the SGI Challenge system used a 256-bit, packet-oriented bus supporting up to 8 processor boards and 32 processors) and later the use of multiple buses and crossbar interconnects—for example, in the Sun SPARCCenter and Enterprise systems (Charlesworth [1998] discussed the interconnect architecture of these multiprocessors). In 2001, the Sun Enterprise servers represented the primary example of large-scale (>16 processors), symmetric multiprocessors in active use. Today, most bus-based machines offer only four or so processors and switches, or alternative designs are used for eight or more.

Toward Large-Scale Multiprocessors

In the effort to build large-scale multiprocessors, two different directions were explored: message-passing multicomputers and scalable shared-memory multiprocessors. Although there had been many attempts to build mesh and hypercube-connected multiprocessors, one of the first multiprocessors to successfully bring together all the pieces was the Cosmic Cube built at Caltech [Seitz 1985]. It introduced important advances in routing and interconnect technology and substantially reduced the cost of the interconnect, which helped make the multicomputer viable. The Intel iPSC 860, a hypercube-connected collection of i860s, was based on these ideas. More recent multiprocessors, such as the Intel Paragon, have used networks with lower dimensionality and higher individual links. The Paragon also employed a separate i860 as a communications controller in each node, although a number of users have found it better to use both i860 processors for computation as well as communication. The Thinking Multiprocessors CM-5 made use of off-the-shelf microprocessors and a fat tree interconnect (see Appendix F). It provided user-level access to the communication channel, thus significantly improving communication latency. In 1995, these two multiprocessors represented the state of the art in message-passing multicomputers.

Early attempts at building a scalable shared-memory multiprocessor include the IBM RP3 [Pfister et al. 1985], the NYU Ultracomputer [Elder et al. 1985; Schwartz 1980], the University of Illinois Cedar project [Gajksi et al. 1983], and the BBN Butterfly and Monarch [BBN Laboratories 1986; Rettberg et al. 1990]. These multiprocessors all provided variations on a nonuniform distributed-memory model and, hence, are distributed shared-memory (DSM) multiprocessors, but they did not support cache coherence, which substantially complicated programming. The RP3 and Ultracomputer projects both explored new ideas in synchronization (fetch-and-operate) as well as the idea of combining references in the network. In all four multiprocessors, the interconnect networks turned out to be more costly than the processing nodes, raising problems for smaller versions of the multiprocessor. The Cray T3D/E (see Arpaci et al. [1995] for an evaluation of the T3D and Scott [1996] for a description of the T3E enhancements) builds on these ideas, using a noncoherent shared address space but building on the advances in interconnect technology developed in the multicomputer domain (see Scott and Thorson [1996]).

Extending the shared-memory model with scalable cache coherence was done by combining a number of ideas. Directory-based techniques for cache coherence were actually known before snooping cache techniques. In fact, the first cache coherence protocols actually used directories, as described by Tang [1976] and implemented in the IBM 3081. Censier and Feautrier [1978] described a directory coherence scheme with tags in memory. The idea of distributing directories with the memories to obtain a scalable implementation of cache coherence was first described by Agarwal et al. [1988] and served as the basis for the Stanford DASH multiprocessor (see Lenoski et al. [1990, 1992]), which was the first operational cache-coherent DSM multiprocessor. DASH was a "plump" node cc-NUMA machine that used four-processor SMPs as its nodes, interconnecting them in a style similar to that of Wildfire but using a more scalable two-dimensional grid rather than a crossbar for the interconnect.

The Kendall Square Research KSR-1 [Burkhardt et al. 1992] was the first commercial implementation of scalable coherent shared memory. It extended the basic DSM approach to implement a concept called *cache-only memory architecture* (COMA), which makes the main memory a cache. In the KSR-1, memory blocks could be replicated in the main memories of each node with hardware support to handle the additional coherence requirements for these replicated blocks. (The KSR-1 was not strictly a pure COMA because it did not migrate the home location of a data item but always kept a copy at home. Essentially, it implemented only replication.) Many other research proposals [Falsafi and Wood 1997; Hagersten, Landin, and Haridi 1992; Saulsbury et al. 1995; Stenström, Joe, and Gupta 1992] for COMA-style architectures and similar approaches that reduce the burden of nonuniform memory access through migration [Chandra et al. 1994; Soundararajan et al. 1998] were developed, but there have been no further commercial implementations.

The Convex Exemplar implemented scalable coherent shared memory using a two-level architecture: At the lowest level, eight-processor modules are built using a crossbar. A ring can then connect up to 32 of these modules, for a total of 256 processors (see Thekkath et al. [1997] for an evaluation). Laudon and Lenoski [1997] described the SGI Origin, which was first delivered in 1996 and is closely based on the original Stanford DASH machine, although including a number of innovations for scalability and ease of programming. Origin uses a bit vector for the directory structure, which is either 16 or 32 bits long. Each bit represents a node, which consists of two processors; a coarse bit vector representation allows each bit to represent up to 8 nodes for a total of 1024 processors. As Galles [1996] described, a high-performance fat hypercube is used for the global interconnect. Hristea, Lenoski, and Keen [1997] have provided a thorough evaluation of the performance of the Origin memory system.

Several research prototypes were undertaken to explore scalable coherence with and without multithreading. These include the MIT Alewife machine [Agarwal et al. 1995] and the Stanford FLASH multiprocessor [Gibson et al. 2000; Kuskin et al. 1994].

Clusters

Clusters were probably “invented” in the 1960s by customers who could not fit all their work on one computer or who needed a backup machine in case of failure of the primary machine [Pfister 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAX clusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users log onto the cluster and are unaware of which machine they are running on. DEC (now HP) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now HP) and IBM (still IBM). Today, virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit.

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf project started with the goal of fulfilling NASA’s desire for a 1 GFLOPS computer for under $50,000. In 1994, a 16-node cluster built from off-the-shelf

PCs using 80486s achieved that goal [Bell and Gray 2001]. This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low-latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at the University of California–Berkeley, connected by 160 MB/sec per link Myrinet switches, was used to set world records in database sort—sorting 8.6 GB of data originally on disk in 1 minute—and in cracking an encrypted message—taking just 3.5 hours to decipher a 40-bit DES key.

This research project, called Network of Workstations [Anderson, Culler, and Patterson 1995], also developed the Inktomi search engine, which led to a startup company with the same name. Google followed the example of Inktomi to build search engines from clusters of desktop computers rather large-scale SMPs, which was the strategy of the leading search engine Alta Vista that Google overtook [Brin and Page 1998]. In 2011, nearly all Internet services rely on clusters to serve their millions of customers.

Clusters are also very popular with scientists. One reason is their low cost, so individual scientists or small groups can own a cluster dedicated to their programs. Such clusters can get results faster than waiting in the long job queues of the shared MPPs at supercomputer centers, which can stretch to weeks. For those interested in learning more, Pfister [1998] wrote an entertaining book on clusters.

Recent Trends in Large-Scale Multiprocessors

In the mid- to late 1990s, it became clear that the hoped for growth in the market for ultralarge-scale parallel computing was unlikely to occur. Without this market growth, it became increasingly clear that the high-end parallel computing market could not support the costs of highly customized hardware and software designed for a small market. Perhaps the most important trend to come out of this observation was that clustering would be used to reach the highest levels of performance. There are now four general classes of large-scale multiprocessors:

- Clusters that integrate standard desktop motherboards using interconnection technology such as Myrinet or Infiniband.
- Multicomputers built from standard microprocessors configured into processing elements and connected with a custom interconnect. These include the Cray XT3 (which used an earlier version of Cray interconnect with a simple cluster architecture) and IBM Blue Gene (more on this unique machine momentarily).
- Clusters of small-scale shared-memory computers, possibly with vector support, which includes the Earth Simulator (which has its own journal available online).

- Large-scale shared-memory multiprocessors, such as the Cray X1 [Dunigan et al. 2005] and SGI Origin and Altix systems. The SGI systems have also been configured into clusters to provide more than 512 processors, although only message passing is supported across the clusters.

The IBM Blue Gene is the most interesting of these designs since its rationale parallels the underlying causes of the recent trend toward multicore in uniprocessor architectures. Blue Gene started as a research project within IBM aimed at the protein sequencing and folding problem. The Blue Gene designers observed that power was becoming an increasing concern in large-scale multiprocessors and that the performance/watt of processors from the embedded space was much better that those in the high-end uniprocessor space. If parallelism was the route to high performance, why not start with the most efficient building block and simply have more of them?

Thus, Blue Gene is constructed using a custom chip that includes an embedded PowerPC microprocessor offering half the performance of a high-end PowerPC, but at a much smaller fraction of the area of power. This allows more system functions, including the global interconnect, to be integrated onto the same die. The result is a highly replicable and efficient building block, allowing Blue Gene to reach much larger processor counts more efficiently. Instead of using stand-alone microprocessors or standard desktop boards as building blocks, Blue Gene uses processor cores. There is no doubt that such an approach provides much greater efficiency. Whether the market can support the cost of a customized design and special software remains an open question.

In 2006, a Blue Gene processor at Lawrence Livermore with 32K processors (and scheduled to go to 65K in late 2005) holds a factor of 2.6 lead in Linpack performance over the third-place system consisting of 20 SGI Altix 512-processor systems interconnected with Infiniband as a cluster.

Blue Gene's predecessor was an experimental machine, QCDOD, which pioneered the concept of a machine using a lower-power embedded microprocessor and tightly integrated interconnect to drive down the cost and power consumption of a node.

Developments in Synchronization and Consistency Models

A wide variety of synchronization primitives have been proposed for shared-memory multiprocessors. Mellor-Crummey and Scott [1991] provided an overview of the issues as well as efficient implementations of important primitives, such as locks and barriers. An extensive bibliography supplies references to other important contributions, including developments in spin locks, queuing locks, and barriers. Lamport [1979] introduced the concept of sequential consistency and what correct execution of parallel programs means. Dubois, Scheurich, and Briggs [1988] introduced the idea of weak ordering (originally in 1986). In 1990, Adve and Hill provided a better definition of weak ordering and also defined the concept of data-race-free; at the same conference, Gharachorloo and his colleagues [1990] introduced release consistency and provided the first data on the performance of relaxed consistency models. More relaxed consistency models have been widely adopted in microprocessor architectures, including the Sun SPARC, Alpha, and IA-64. Adve and Gharachorloo [1996] have provided an excellent tutorial on memory consistency and the differences among these models.

Other References

The concept of using virtual memory to implement a shared address space among distinct machines was pioneered in Kai Li's Ivy system in 1988. There have been subsequent papers exploring hardware support issues, software mechanisms, and programming issues. Amza et al. [1996] described a system built on workstations using a new consistency model, Kontothanassis et al. [1997] described a software shared-memory scheme using remote writes, and Erlichson et al. [1996] described the use of shared virtual memory to build large-scale multiprocessors using SMPs as nodes.

There is an almost unbounded amount of information on multiprocessors and multicomputers: Conferences, journal papers, and even books seem to appear faster than any single person can absorb the ideas. No doubt many of these papers will go unnoticed—not unlike the past. Most of the major architecture conferences contain papers on multiprocessors. An annual conference, Supercomputing *XY* (where *X* and *Y* are the last two digits of the year), brings together users, architects, software developers, and vendors, and the proceedings are published in book, CD-ROM, and online (see *www.scXY.org*) form. Two major journals, *Journal of Parallel and Distributed Computing* and the *IEEE Transactions on Parallel and Distributed Systems*, contain papers on all aspects of parallel processing. Several books focusing on parallel processing are included in the following references, with Culler, Singh, and Gupta [1999] being the most recent, large-scale effort. For years, Eugene Miya of NASA Ames Research Center has collected an online bibliography of parallel-processing papers. The bibliography, which now contains more than 35,000 entries, is available online at *liinwww.ira.uka.de/bibliography/Parallel/Eugene/index.html*.

In addition to documenting the discovery of concepts now used in practice, these references also provide descriptions of many ideas that have been explored and found wanting, as well as ideas whose time has just not yet come. Given the move toward multicore and multiprocessors as the future of high-performance computer architecture, we expect that many new approaches will be explored in the years ahead. A few of them will manage to solve the hardware and software problems that have been the key to using multiprocessing for the past 40 years!

References

Adve, S. V., and K. Gharachorloo [1996]. "Shared memory consistency models: A tutorial," *IEEE Computer* 29:12 (December), 66–76.

Adve, S. V., and M. D. Hill [1990]. "Weak ordering—a new definition," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 2–14.

Agarwal, A., R. Bianchini, D. Chaiken, K. Johnson, and D. Kranz [1995]. "The MIT Alewife machine: Architecture and performance," *22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 2–13.

Agarwal, A., J. L. Hennessy, R. Simoni, and M. A. Horowitz [1988]. "An evaluation of directory schemes for cache coherence," *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, May 30–June 2, 1988, Honolulu, Hawaii, 280–289.

Agarwal, A., J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin [1993]. "Sparcle: An evolutionary processor design for large-scale multiprocessors," *IEEE Micro* 13 (June), 48–61.

Alles, A. [1995]. "ATM Internetworking," White Paper (May), Cisco Systems, Inc., San Jose, Calif. (*www.cisco.com/warp/public/614/12.html*).

Almasi, G. S., and A. Gottlieb [1989]. *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, Calif.

Alverson, G., R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith [1992]. "Exploiting heterogeneous parallelism on a multithreaded multiprocessor," *Proc. ACM/IEEE Conf. on Supercomputing*, November 16–20, 1992, Minneapolis, Minn., 188–197.

Amdahl, G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, April 18–20, 1967, Atlantic City, N.J., 483–485.

Amza C., A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel [1996]. "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer* 29:2 (February), 18–28.

Anderson, T. E., D. E. Culler, and D. Patterson [1995]. "A case for NOW (networks of workstations)," *IEEE Micro* 15:1 (February), 54–64.

Ang, B., D. Chiou, D. Rosenband, M. Ehrlich, L. Rudolph, and Arvind [1998]. "StarT-Voyager: A flexible platform for exploring scalable SMP issues," *Proc. ACM/IEEE Conf. on Supercomputing*, November 7–13, 1998, Orlando, FL.

Archibald, J., and J.-L. Baer [1986]. "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. on Computer Systems* 4:4 (November), 273–298.

Arpaci, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick [1995]. "Empirical evaluation of the CRAY-T3D: A compiler perspective," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy.

Baer, J.-L., and W.-H. Wang [1988]. "On the inclusion properties for multi-level cache hierarchies," *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, May 30–June 2, 1988, Honolulu, Hawaii, 73–80.

Balakrishnan, H. V., N. Padmanabhan, S. Seshan, and R. H. Katz [1997]. "A comparison of mechanisms for improving TCP performance over wireless links," *IEEE/ACM Trans. on Networking* 5:6 (December), 756–769.

Barroso, L. A., K. Gharachorloo, and E. Bugnion [1998]. "Memory system characterization of commercial workloads," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 3–14.

Baskett, F., T. Jermoluk, and D. Solomon [1988]. “The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 10,000 lighted polygons per second,” *Proc. IEEE COMPCON*, February 29–March 4, 1988, San Francisco, 468–471.

BBN Laboratories. [1986]. *Butterfly Parallel Processor Overview*, Tech. Rep. 6148, BBN Laboratories, Cambridge, Mass.

Bell, C. G. [1985]. “Multis: A new class of multiprocessor computers,” *Science* 228 (April 26), 462–467.

Bell, C. G. [1989]. “The future of high performance computers in science and engineering,” *Communications of the ACM* 32:9 (September), 1091–1101.

Bell, C. G., and J. Gray [2001]. *Crays, Clusters and Centers*, Tech. Rep. MSR-TR-2001-76, Microsoft Research, Redmond, Wash.

Bell, C. G., and J. Gray [2002]. “What’s next in high performance computing,” *CACM*, 45:2 (February), 91–95.

Bouknight, W. J., S. A. Deneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick [1972]. “The Illiac IV system,” *Proc. IEEE* 60:4, 369–379. Also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982, 306–316.

Brain, M. [2000]. *Inside a Digital Cell Phone, www.howstuffworks.com/inside-cell-phone.htm*.

Brewer, E. A., and B. C. Kuszmaul [1994]. “How to get good performance from the CM-5 data network,” *Proc. Eighth Int’l. Parallel Processing Symposium (IPPS)*, April 26–29, 1994, Cancun, Mexico.

Brin, S., and L. Page [1998]. “The anatomy of a large-scale hypertextual Web search engine,” *Proc. 7th Int’l. World Wide Web Conf.*, April 14–18, 1998, Brisbane, Queensland, Australia, 107–117.

Burkhardt III, H., S. Frank, B. Knobe, and J. Rothnie [1992]. *Overview of the KSR1 Computer System*, Tech. Rep. KSR-TR-9202001, Kendall Square Research, Boston.

Censier, L., and P. Feautrier [1978]. “A new solution to coherence problems in multicache systems,” *IEEE Trans. on Computers* C-27:12 (December), 1112–1118.

Chandra, R., S. Devine, B. Verghese, A. Gupta, and M. Rosenblum [1994]. “Scheduling and page migration for multiprocessor compute servers,” *Proc. Sixth Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, 1994, San Jose, Calif., 12–24.

Charlesworth, A. [1998]. “Starfire: Extending the SMP envelope,” *IEEE Micro* 18:1 (January/February), 39–49.

Clark, W. A. [1957]. “The Lincoln TX-2 computer development,” *Proc. Western Joint Computer Conference*, February 26–28, 1957, Los Angeles, 143–145.

Comer, D. [1993]. *Internetworking with TCP/IP*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Culler, D. E., J. P. Singh, and A. Gupta [1999]. *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco.

Dally, W. J., and C. I. Seitz [1986]. “The torus routing chip,” *Distributed Computing* 1:4, 187–196.

Davie, B. S., L. L. Peterson, and D. Clark [1999]. *Computer Networks: A Systems Approach*, 2nd ed., Morgan Kaufmann, San Francisco.

Desurvire, E. [1992]. "Lightwave communications: The fifth generation," *Scientific American* (International Edition) 266:1 (January), 96–103.

Dongarra, J., T. Sterling, H. Simon, and E. Strohmaier [2005]. "High-performance computing: Clusters, constellations, MPPs, and future directions," *Computing in Science & Engineering*, 7:2 (March/April), 51–59.

Dubois, M., C. Scheurich, and F. Briggs [1988]. "Synchronization, coherence, and event ordering," *IEEE Computer* 21:2 (February), 9–21.

Dunigan, W., K. Vetter, K. White, and P. Worley [2005]. "Performance evaluation of the Cray X1 distributed shared memory architecture," *IEEE Micro*, January/February, 30–40.

Eggers, S. [1989]. "Simulation Analysis of Data Sharing in Shared Memory Multiprocessors," Ph.D. thesis, Computer Science Division, University of California, Berkeley.

Elder, J., A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Randolph, M. Snir, P. Teller, and J. Wilson [1985]. "Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach," *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass., 126–135.

Erlichson, A., N. Nuckolls, G. Chesson, and J. L. Hennessy [1996]. "SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory," *Proc. Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass., 210–220.

Falsafi, B., and D. A. Wood [1997]. "Reactive NUMA: A design for unifying S-COMA and CC-NUMA," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 229–240.

Flynn, M. J. [1966]. "Very high-speed computing systems," *Proc. IEEE* 54:12 (December), 1901–1909.

Forgie, J. W. [1957]. "The Lincoln TX-2 input-output system," *Proc. Western Joint Computer Conference*, February 26–28, 1957, Los Angeles, 156–160.

Frank, S. J. [1984]. "Tightly coupled multiprocessor systems speed memory access time," *Electronics* 57:1 (January), 164–169.

Gajski, D., D. Kuck, D. Lawrie, and A. Sameh [1983]. "CEDAR—a large scale multiprocessor," *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, August, Columbus, Ohio, 524–529.

Galles, M. [1996]. "Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip," *Proc. IEEE HOT Interconnects '96*, August 15–17, 1996, Stanford University, Palo Alto, Calif.

Gehringer, E. F., D. P. Siewiorek, and Z. Segall [1987]. *Parallel Processing: The Cm* Experience*, Digital Press, Bedford, Mass.

Gharachorloo, K., A. Gupta, and J. L. Hennessy [1992]. "Hiding memory latency using dynamic scheduling in shared-memory multiprocessors," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy [1990]. “Memory consistency and event ordering in scalable shared-memory multiprocessors,” *Proc. 17th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 15–26.

Gibson, J., R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich [2000]. “FLASH vs. (simulated) FLASH: Closing the simulation loop,” *Proc. Ninth Int’l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 12–15, Cambridge, Mass., 49–58.

Goodman, J. R. [1983]. “Using cache memory to reduce processor memory traffic,” *Proc. 10th Annual Int’l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 124–131.

Goralski, W. [1997]. *SONET: A Guide to Synchronous Optical Network*, McGraw-Hill, New York.

Grice, C., and M. Kanellos [2000]. “Cell phone industry at crossroads: Go high or low?” *CNET News* (August 31), *technews.netscape.com/news/0-1004-201-2518386-0.html?tag=st.ne.1002.tgif.sf*.

Groe, J. B., and L. E. Larson [2000]. *CDMA Mobile Radio Design*, Artech House, Boston.

Hagersten E., and M. Koster [1998]. “WildFire: A scalable path for SMPs,” *Proc. Fifth Int’l. Symposium on High-Performance Computer Architecture*, January 9–12, 1999, Orlando, Fla.

Hagersten, E., A. Landin, and S. Haridi [1992]. “DDM—a cache-only memory architecture,” *IEEE Computer* 25:9 (September), 44–54.

Hill, M. D. [1998]. “Multiprocessors should support simple memory consistency models,” *IEEE Computer* 31:8 (August), 28–34.

Hillis, W. D. [1985]. *The Connection Multiprocessor*, MIT Press, Cambridge, Mass.

Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa [1992]. “An elementary processor architecture with simultaneous instruction issuing from multiple threads,” *Proc. 19th Annual Int’l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 136–145.

Hockney, R. W., and C. R. Jesshope [1988]. *Parallel Computers 2: Architectures, Programming and Algorithms*, Adam Hilger, Ltd., Bristol, England.

Holland, J. H. [1959]. “A universal computer capable of executing an arbitrary number of subprograms simultaneously,” *Proc. East Joint Computer Conf.* 16, 108–113.

Hord, R. M. [1982]. *The Illiac-IV, The First Supercomputer*, Computer Science Press, Rockville, Md.

Hristea, C., D. Lenoski, and J. Keen [1997]. “Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks,” *Proc. ACM/IEEE Conf. on Supercomputing*, November 15–21, 1997, San Jose, Calif.

Hwang, K. [1993]. *Advanced Computer Architecture and Parallel Programming*, McGraw-Hill, New York.

IBM. [2005]. "Blue Gene," *IBM J. of Research and Development*, 49:2/3 (special issue).

Infiniband Trade Association. [2001]. *InfiniBand Architecture Specifications Release 1.0.a, www.infinibandta.org*.

Jordan, H. F. [1983]. "Performance measurements on HEP—a pipelined MIMD computer," *Proc. 10th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 5–7, 1982, Stockholm, Sweden, 207–212.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397–1407.

Keckler, S. W., and W. J. Dally [1992]. "Processor coupling: Integrating compile time and runtime scheduling for parallelism," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 202–213.

Kontothanassis, L., G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott [1997]. "VM-based shared memory on low-latency, remotememory-access networks," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo.

Kurose, J. F., and K. W. Ross [2001]. *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley, Boston.

Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy [1994]. "The Stanford FLASH multiprocessor," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago.

Lamport, L. [1979]. "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers* C-28:9 (September), 241–248.

Laudon, J., A. Gupta, and M. Horowitz [1994]. "Interleaving: A multithreading technique targeting multiprocessors and workstations," *Proc. Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 4–7, 1994, San Jose, Calif., 308–318.

Laudon, J., and D. Lenoski [1997]. "The SGI Origin: A ccNUMA highly scalable server," *Proc. 24th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–4, 1997, Denver, Colo., 241–251.

Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy [1990]. "The Stanford DASH multiprocessor," *Proc. 17th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 28–31, 1990, Seattle, Wash., 148–159.

Lenoski, D., J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. Lam [1992]. "The Stanford DASH multiprocessor," *IEEE Computer* 25:3 (March), 63–79.

Li, K. [1988]. "IVY: A shared virtual memory system for parallel computing," *Proc. Int'l. Conf. on Parallel Processing (ICCP)*, August, The Pennsylvania State University, University Park, Penn.

Lo, J., L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh [1998]. "An analysis of database workload performance on simultaneous multithreaded processors," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 39–50.

Lo, J., S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen [1997]. "Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading," *ACM Trans. on Computer Systems* 15:2 (August), 322–354.

Lovett, T., and S. Thakkar [1988]. "The Symmetry multiprocessor system," *Proc. Int'l. Conf. on Parallel Processing (ICCP)*, August, The Pennsylvania State University, University Park, Penn., 303–310.

Mellor-Crummey, J. M., and M. L. Scott [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems* 9:1 (February), 21–65.

Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," *Bibliothèque Universelle de Genève*, 82 (October).

Metcalfe, R. M. [1993]. "Computer/network interface design: Lessons from Arpanet and Ethernet." *IEEE J. on Selected Areas in Communications* 11:2 (February), 173–180.

Metcalfe, R. M., and D. R. Boggs [1976]. "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM* 19:7 (July), 395–404.

Mitchell, D. [1989]. "The Transputer: The time is now," *Computer Design* (RISC suppl.), 40–41.

Miya, E. N. [1985]. "Multiprocessor/distributed processing bibliography," *Computer Architecture News* 13:1, 27–29.

National Research Council. [1997]. *The Evolution of Untethered Communications*, Computer Science and Telecommunications Board, National Academy Press, Washington, D.C.

Nikhil, R. S., G. M. Papadopoulos, and Arvind [1992]. "*T: A multithreaded massively parallel architecture," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 156–167.

Noordergraaf, L., and R. van der Pas [1999]. "Performance experiences on Sun's WildFire prototype," *Proc. ACM/IEEE Conf. on Supercomputing*, November 13–19, 1999, Portland, Ore.

Partridge, C. [1994]. *Gigabit Networking*, Addison-Wesley, Reading, Mass.

Pfister, G. F. [1998]. *In Search of Clusters*, 2nd ed., Prentice Hall, Upper Saddle River, N.J.

Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfekder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss [1985]. "The IBM research parallel processor prototype (RP3): Introduction and architecture," *Proc. 12th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–19, 1985, Boston, Mass., 764–771.

Reinhardt, S. K., J. R. Larus, and D. A. Wood [1994]. "Tempest and Typhoon: User-level shared memory," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 325–336.

Rettberg, R. D., W. R. Crowther, P. P. Carvey, and R. S. Towlinson [1990]. "The Monarch parallel processor hardware design," *IEEE Computer* 23:4 (April), 18–30.

Rosenblum, M., S. A. Herrod, E. Witchel, and A. Gupta [1995]. "Complete computer simulation: The SimOS approach," in *IEEE Parallel and Distributed Technology* (now called *Concurrency*) 4:3, 34–43.

Saltzer, J. H., D. P. Reed, and D. D. Clark [1984]. "End-to-end arguments in system design," *ACM Trans. on Computer Systems* 2:4 (November), 277–288.

Satran, J., D. Smith, K. Meth, C. Sapuntzakis, M. Wakeley, P. Von Stamwitz, R. Haagens, E. Zeidner, L. Dalle Ore, and Y. Klein [2001]. "iSCSI," IPS Working Group of IETF, Internet draft *www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt*.

Saulsbury, A., T. Wilkinson, J. Carter, and A. Landin [1995]. "An argument for Simple COMA," *Proc. First IEEE Symposium on High-Performance Computer Architectures*, January 22–25, 1995, Raleigh, N.C., 276–285.

Schwartz, J. T. [1980]. "Ultracomputers," *ACM Trans. on Programming Languages and Systems* 4:2, 484–521.

Scott, S. L. [1996]. "Synchronization and communication in the T3E multiprocessor," *Seventh Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1–5, 1996, Cambridge, Mass., 26–36.

Scott, S. L., and G. M. Thorson [1996]. "The Cray T3E network: Adaptive routing in a high-performance 3D torus," *Proc. IEEE HOT Interconnects '96*, August 15–17, 1996, Stanford University, Palo Alto, Calif., 14–156.

Seitz, C. L. [1985]. "The Cosmic Cube (concurrent computing)," *Communications of the ACM* 28:1 (January), 22–33.

Singh, J. P., J. L. Hennessy, and A. Gupta [1993]. "Scaling parallel programs for multiprocessors: Methodology and examples," *Computer* 26:7 (July), 22–33.

Slotnick, D. L., W. C. Borck, and R. C. McReynolds [1962]. "The Solomon computer," *Proc. AFIPS Fall Joint Computer Conf.*, December 4–6, 1962, Philadelphia, Penn., 97–107.

Smith, B. J. [1978]. "A pipelined, shared resource MIMD computer," *Proc. Int'l. Conf. on Parallel Processing (ICPP)*, August, Bellaire, Mich., 6–8.

Soundararajan, V., M. Heinrich, B. Verghese, K. Gharachorloo, A. Gupta, and J. L. Hennessy [1998]. "Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors," *Proc. 25th Annual Int'l. Symposium on Computer Architecture (ISCA)*, July 3–14, 1998, Barcelona, Spain, 342–355.

Spurgeon, C. [2001]. "Charles Spurgeon's Ethernet Web site," *www.host.ots.utexas.edu/ethernet/ethernet-home.html*.

Stenström, P., T. Joe, and A. Gupta [1992]. "Comparative performance evaluation of cachecoherent NUMA and COMA architectures," *Proc. 19th Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 19–21, 1992, Gold Coast, Australia, 80–91.

Sterling, T. [2001]. *Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux*, MIT Press, Cambridge, Mass.

Stevens, W. R. [1994–1996]. *TCP/IP Illustrated* (three volumes), Addison-Wesley, Reading, Mass.

Stone, H. [1991]. *High Performance Computers*, Addison-Wesley, New York.

Swan, R. J., A. Bechtolsheim, K. W. Lai, and J. K. Ousterhout [1977]. "The implementation of the Cm* multi-microprocessor," *Proc. AFIPS National Computing Conf.*, June 13–16, 1977, Dallas, Tex., 645–654.

Swan, R. J., S. H. Fuller, and D. P. Siewiorek [1977]. "Cm*—a modular, multi-microprocessor," *Proc. AFIPS National Computing Conf.*, June 13–16, 1977, Dallas, Tex., 637–644.

Tanenbaum, A. S. [1988]. *Computer Networks*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

Tang, C. K. [1976]. "Cache design in the tightly coupled multiprocessor system," *Proc. AFIPS National Computer Conf.*, June 7–10, 1976, New York, 749–753.

Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs [1982]. "Alto: A personal computer," in D. P. Siewiorek, C. G. Bell, and A. Newell, eds., *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 549–572.

Thekkath, R., A. P. Singh, J. P. Singh, S. John, and J. L. Hennessy [1997]. "An evaluation of a commercial CC-NUMA architecture—the CONVEX Exemplar SPP1200," *Proc. 11th Int'l. Parallel Processing Symposium (IPPS)*, April 1–7, 1997, Geneva, Switzerland.

Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm [1996]. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. 23rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, May 22–24, 1996, Philadelphia, Penn., 191–202.

Tullsen, D. M., S. J. Eggers, and H. M. Levy [1995]. "Simultaneous multithreading: Maximizing on-chip parallelism," *Proc. 22nd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 22–24, 1995, Santa Margherita, Italy, 392–403.

Unger, S. H. [1958]. "A computer oriented towards spatial problems," *Proc. Institute of Radio Engineers* 46:10 (October), 1744–1750.

Walrand, J. [1991]. *Communication Networks: A First Course*, Aksen Associates: Irwin, Homewood, Ill.

Wilson, A. W., Jr. [1987]. "Hierarchical cache/bus architecture for shared-memory multiprocessors," *Proc. 14th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 2–5, 1987, Pittsburgh, Penn., 244–252.

Wolfe, A., and J. P. Shen [1991]. "A variable instruction stream extension to the VLIW architecture." *Proc. Fourth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 8–11, 1991, Palo Alto, Calif., 2–14.

Wood, D. A., and M. D. Hill [1995]. "Cost-effective parallel computing," *IEEE Computer* 28:2 (February), 69–72.

Wulf, W., and C. G. Bell [1972]. "C.mmp—A multi-mini-processor," *Proc. AFIPS Fall Joint Computer Conf.*, December 5–7, 1972, Anaheim, Calif., 765–777.

Wulf, W., and S. P. Harbison [1978]. "Reflections in a pool of processors—an experience report on C.mmp/Hydra," *Proc. AFIPS National Computing Conf.* June 5–8, 1978, Anaheim, Calif., 939–951.

Yamamoto, W., M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky [1994]. “Performance estimation of multistreamed, superscalar processors,” *Proc. 27th Hawaii Int’l. Conf. on System Sciences*, January 4–7, 1994, Wailea, 195–204.

M.8 The Development of Clusters (Chapter 6)

In this section, we cover the development of clusters that were the foundation of warehouse-scale computers (WSCs) and of utility computing. (Readers interested in learning more should start with Barroso and Hölzle [2009] and the blog postings and talks of James Hamilton at *http://perspectives.mvdirona.com*.)

Clusters, the Forerunner of WSCs

Clusters were probably “invented” in the 1960s by customers who could not fit all their work on one computer or who needed a backup machine in case of failure of the primary machine [Pfister 1998]. Tandem introduced a 16-node cluster in 1975. Digital followed with VAX clusters, introduced in 1984. They were originally independent computers that shared I/O devices, requiring a distributed operating system to coordinate activity. Soon they had communication links between computers, in part so that the computers could be geographically distributed to increase availability in case of a disaster at a single site. Users log onto the cluster and are unaware of which machine they are running on. DEC (now HP) sold more than 25,000 clusters by 1993. Other early companies were Tandem (now HP) and IBM (still IBM). Today, virtually every company has cluster products. Most of these products are aimed at availability, with performance scaling as a secondary benefit.

Scientific computing on clusters emerged as a competitor to MPPs. In 1993, the Beowulf project started with the goal of fulfilling NASA’s desire for a 1 GFLOPS computer for under $50,000. In 1994, a 16-node cluster built from off-the-shelf PCs using 80486s achieved that goal [Bell and Gray 2001]. This emphasis led to a variety of software interfaces to make it easier to submit, coordinate, and debug large programs or a large number of independent programs.

Efforts were made to reduce latency of communication in clusters as well as to increase bandwidth, and several research projects worked on that problem. (One commercial result of the low-latency research was the VI interface standard, which has been embraced by Infiniband, discussed below.) Low latency then proved useful in other applications. For example, in 1997 a cluster of 100 UltraSPARC desktop computers at the University of California–Berkeley, connected by 160 MB/sec per link Myrinet switches, was used to set world records in database sort—sorting 8.6 GB of data originally on disk in 1 minute—and in cracking an encrypted message—taking just 3.5 hours to decipher a 40-bit DES key.

This research project, called Network of Workstations [Anderson, Culler, and Patterson 1995], also developed the Inktomi search engine, which led to a start-up company with the same name. Eric Brewer led the Inktomi effort at Berkeley and then at the company to demonstrate the use of commodity hardware to build computing infrastructure for Internet services. Using standardized networks within a rack of PC servers gave Inktomi better scalability. In contrast, the strategy of the prior leading search engine Alta Vista was to build from large-scale SMPs. Compared to the high-performance computing work in clusters, the emphasis was on a relatively large number of low-cost nodes and a clear programming model. Hence, the NOW project and Inktomi are considered the foundation of WSCs and Cloud Computing. Google followed the example of Inktomi technology when it took the leading search engine mantle from Inktomi just as Inktomi had taken it from Alta Vista [Brin and Page 1998]. (Google's initial innovation was search quality; the WSC innovations came much later.) For many years now, all Internet services have relied on cluster technology to serve their millions of customers.

Utility Computing, the Forerunner of Cloud Computing

As stated in the text, the earliest version of utility computing was timesharing. Although timesharing faded away over time with the creation of smaller and cheaper personal computers, in the last decade there have been many less than fully successful attempts to resuscitate utility computing. Sun began selling time on Sun Cloud at $1 per hour in 2000, HP offered a Utility Data Center in 2001, and Intel tried selling time on internal supercomputers in the early 2000s. Although they were commercially available, few customers used them.

A related topic is *grid computing*, which was originally invented so that scientific programs could be run across geographically distributed computing facilities. At the time, some questioned the wisdom of this goal, setting aside how difficult it would be to achieve. Grid computing tended to require very large systems running very large programs, using multiple datacenters for the tasks. Single applications did not really run well when geographically distributed, given the long latencies inherent with long distance. This first step eventually led to some conventions for data access, but the grid computing community did not develop APIs that were useful beyond the high-performance computing community, so the cloud computing effort shares little code or history with grid computing.

Armbrust et al [2009] argued that, once the Internet service companies solved the operational problems to work at large scale, the significant economies of scale that they uncovered brought their costs down below those of smaller datacenters. Amazon recognized that if this cost advantage was true then Amazon should be able to make a profit selling this service. In 2006, Amazon announced Elastic Cloud Computing (EC2) at $0.10 per hour per instance. The subsequent popularity of EC2 led other Internet companies to offer cloud computing services, such as Google App Engine and Microsoft Azure, albeit at higher abstraction levels than the x86 virtual machines of Amazon Web Services. Hence, the current popularity of pay-as-you go computing isn't because someone recently came up with the idea;

it's because the technology and business models have aligned so that companies can make money offering a service that many people want to use. Time will tell whether there will be many successful utility computing models or whether the industry will converge around a single standard. It will certainly be interesting to watch.

Containers

In the fall of 2003, many people were thinking about using containers to hold servers. Brewster Kahle, director and founder of the Internet Archive, gave talks about how he could fit the whole archive in a single 40-foot container. His interest was making copies of the Archive and distributing it around the world to ensure its survivability, thereby avoiding the fate of the Library of Alexandria that was destroyed by fire in 48 B.C.E. People working with Kahle wrote a white paper based on his talk in November 2003 to get more detail about what a container design would look like.

That same year, engineers at Google were also looking at building datacenters using containers and submitted a patent on aspects of it in December 2003. The first container for a datacenter was delivered in January 2005, and Google received the patent in October 2007. Google publicly revealed the use of containers in April 2009.

Greg Papadopolous of Sun Microsystems and Danny Hillis of Applied Minds heard Kahle's talk and designed a product called the Sun Modular Datacenter that debuted in October 2006. (The project code name was Black Box, a term many people still use.) This half-length (20-foot) container could hold 280 servers. This product release combined with Microsoft's announcement that they were building a datacenter designed to hold 220 40-foot containers inspired many other companies to offer containers and servers designed to be placed in them.

In a nice turn of events, in 2009 the Internet Archive migrated its data to a Sun Modular Datacenter. A copy of the Internet Archive is now at the New Library of Alexandria in Egypt, near the site of the original library.

References

Anderson, T. E., D. E. Culler, and D. Patterson [1995]. "A case for NOW (networks of workstations)," *IEEE Micro* 15:1 (February), 54–64.

Apache Software Foundation. [2011]. Apache Hadoop project, *http://hadoop.apache.org*.

Armbrust, M., A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia [2009]. *Above the Clouds: A Berkeley View of Cloud Computing*, Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley (http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html).

Barroso, L. A. [2010]. “Warehouse scale computing [keynote address],” *Proc. ACM SIG-MOD*, June 8–10, 2010, Indianapolis, Ind.

Barroso, L. A., and U. Hölzle [2007]. “The case for energy-proportional computing,” *IEEE Computer* 40:12 (December), 33–37.

Barroso, L.A., and U. Hölzle [2009]. “The datacenter as a computer: An introduction to the design of warehouse-scale machines,” in M. D. Hill, ed., *Synthesis Lectures on Computer Architecture*, Morgan & Claypool, San Rafael, Calif.

Barroso, L.A., Clidaras, J. and Hölzle, U., 2013. *The datacenter as a computer:An introduction to the design of warehouse-scale machines*. Synthesis lectures on computer architecture, 8(3), pp.1–154.

Barroso, L.A., Marty, M., Patterson, D., and Ranganathan, P. 2017. Attack of the Killer Microseconds. *Communications of the ACM*, 56(2).

Bell, C. G., and J. Gray [2002]. “What’s next in high performance computing,” *Communications of the ACM* 45:2 (February), 91–95.

Brady, J.T., 1986. A theory of productivity in the creative process. IEEE Computer Graphics and Applications, 6(5), pp.25–34.

Brin, S., and L. Page [1998]. “The anatomy of a large-scale hypertextual Web search engine,” *Proc. 7th Int’l. World Wide Web Conf.*, April 14–18, 1998, Brisbane, Queensland, Australia, 107–117.

Carter, J., and K. Rajamani [2010]. “Designing energy-efficient servers and data centers,” *IEEE Computer* 43:7 (July), 76–78.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber [2006]. “Bigtable: A distributed storage system for structured data,” in *Proc. Operating Systems Design and Implementation (OSDI ’06)*, November 6–8, 2006, Seattle, Wash.

Chang, J., J. Meza, P. Ranganathan, C. Bash, and A. Shah [2010]. “Green server design: Beyond operational energy to sustainability,” *Workshop on Power Aware Computing and Systems (HotPower ’10)*, October 4–6, 2010, Vancouver, British Columbia.

Clark, J., 2014 Five Numbers That Illustrate the Mind-Bending Size of Amazon’s Cloud, Bloomberg, https://www.bloomberg.com/news/2014-11-14/5-numbersthat-illustrate-the-mind-bending-size-of-amazon-s-cloud.html.

Clidaras, J., C. Johnson, and B. Felderman [2010]. Private communication.

Climate Savers Computing. [2007]. Efficiency specs, *http://www.climatesaverscomputing.org/*.

Clos, C., 1953. A Study of Non-Blocking Switching Networks. *Bell Labs Technical Journal*, 32(2), pp.406-424.

Dean, J. [2009]. “Designs, lessons and advice from building large distributed systems [keynote address],” *Proc. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, Co-located with the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, October 10–11, 2009, Big Sky, Mont.

Dean, J. and Barroso, L.A., 2013. The tail at scale. *Communications of the ACM*, 56(2), pp.74-80.

Dean, J., and S. Ghemawat [2004]. “MapReduce: Simplified data processing on large clusters.” In *Proc. Operating Systems Design and Implementation (OSDI ’04)*, December 6–8, 2004, San Francisco, 137–150.

Dean, J., and S. Ghemawat [2008]. “MapReduce: simplified data processing on large clusters,” *Communications of the ACM* 51:1, 107–113.

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels [2007]. “Dynamo: Amazon’s highly available key-value store,” in *Proc. 21st ACM Symposium on Operating Systems Principles*, October 14–17, 2007, Stevenson, Wash.

Doherty, W.J. and Thadhani, A.J., 1982. The economic value of rapid response time. IBM Report.

Fan, X., W. Weber, and L. A. Barroso [2007]. “Power provisioning for a warehouse-sized computer,” in *Proc. 34th Annual Int’l. Symposium on Computer Architecture (ISCA)*, June 9–13, 2007, San Diego, Calif.

A. Fikes, “Storage architecture and challenges,” in Google Faculty Summit, 2010.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003). “The Google file system,” in *Proc. 19th ACM Symposium on Operating Systems Principles*, October 19–22, 2003, Lake George, N.Y.

Greenberg, A., N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta [2009]. “VL2: A scalable and flexible data center network,” in *Proc. SIGCOMM*, August 17–21, Barcelona, Spain.

González, A.and Day, M. April 27, 2016, “Amazon, Microsoft invest billions as computing shifts to cloud,” *The Seattle Times*. http://www.seattletimes.com/business/technology/amazon-microsoft-invest-billions-as-computing-shifts-to-cloud/

Hamilton, J. [2009]. “Data center networks are in my way,” Stanford Clean Slate CTO Summit, October 23, 2009, *http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CleanSlateCTO2009.pdf*.

Hamilton, J. [2010]. “Cloud computing economies of scale,” *Proc. AWS Workshop on Genomics & Cloud Computing*, June 8, 2010, Seattle, Wash. (*http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf*).

Hamilton, J., 2014. AWS Innovation at Scale, AWS Re-invent conference. https://www.youtube.com/watch?v=JIQETrFC_SQ

Hamilton, J., May 2015. The Return to the Cloud, http://perspectives.mvdirona.com/2015/05/the-return-to-the-cloud//

Hamilton, J., April 2017. How Many Data Centers Needed World-Wide, http://perspectives.mvdirona.com/2017/04/how-many-data-centers-needed-worldwide/

Hölzle, U. [2010]. “Brawny cores still beat wimpy cores, most of the time,” *IEEE Micro*, July/August.

Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.Y. and Brooks, D., 2015, June. Profiling a warehouse-scale computer. *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*.

Lang, W., J. M. Patel, and S. Shankar [2010]. "Wimpy node clusters: What about non-wimpy workloads?" *Proc. Sixth Int'l. Workshop on Data Management on New Hardware*, June 7, 2010, Indianapolis, Ind.

Lim, K., P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt [2008]. "Understanding and designing new system architectures for emerging warehouse-computing environments," *Proc. 35th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 21–25, 2008, Beijing, China.

Narayanan, D., E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron [2009]. "Migrating server storage to SSDs: Analysis of trade-offs," *Proc. 4th ACM European Conf. on Computer Systems*, April 1–3, 2009, Nuremberg, Germany.

Pfister, G. F. [1998]. *In Search of Clusters*, 2nd ed., Prentice Hall, Upper Saddle River, N.J.

Pinheiro, E., W.-D. Weber, and L. A. Barroso [2007]. "Failure trends in a large disk drive population," *Proc. 5th USENIX Conference on File and Storage Technologies (FAST '07)*, February 13–16, 2007, San Jose, Calif.

Ranganathan, P., P. Leech, D. Irwin, and J. Chase [2006]. "Ensemble-level power management for dense blade servers," *Proc. 33rd Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 17–21, 2006, Boston, Mass., 66–77.

Reddi, V. J., B. C. Lee, T. Chilimbi, and K. Vaid [2010]. "Web search using mobile cores: Quantifying and mitigating the price of efficiency," *Proc. 37th Annual Int'l. Symposium on Computer Architecture (ISCA)*, June 19–23, 2010, Saint-Malo, France.

Schroeder, B., and G. A. Gibson [2007]. "Understanding failures in petascale computers," *Journal of Physics: Conference Series 78*, 188–198.

Schroeder, B., E. Pinheiro, and W.-D. Weber [2009]. "DRAM errors in the wild: A large-scale field study," *Proc. Eleventh Int'l. Joint Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 15–19, 2009, Seattle, Wash.

Schurman, E. and J. Brutlag [2009]. "The User and Business Impact of Server Delays," *Proc. Velocity: Web Performance and Operations Conf.*, June 22–24, 2009, San Jose, Calif.

Tezzaron Semiconductor. [2004]. "*Soft Errors in Electronic Memory—A White Paper*, Tezzaron Semiconductor, Naperville, Ill. (*http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf*).

Vahdat, A., M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan [2010]. "Scale-out networking in the data center," *IEEE Micro* July/August 2010.

M.9 Historical Perspectives and References

As architects experiment with DSAs, knowing architecture history may help. There are likely older architecture ideas that were unsuccessful for general-purpose computing that could nevertheless make eminent sense for domain-specific architectures. After all, they probably did some things well, and either they might match your domain, or, conversely, your domain might omit features that were challenges for these architectures. For example, both the Illiac IV (Barnes et al., 1968) from the 1960s and the FPS 120a (Charlesworth, 1981) from the 1970s had two-dimensional arrays of processing elements, so they are proper ancestors to the TPU and Paintbox. Similarly, while VLIW architectures of the Multiflow (Rau and Fisher, 1993) and Itanium (Sharangpani and Arora, 2000) were not commercial successes for general-purpose computing, Paintbox does not have the erratic data cache misses, unpredictable branches, or large code footprint that were difficult for VLIW architectures.

Two survey articles document that custom neural network ASICs go back at least 25 years (Ienne et al., 1996; Asanović, 2002). For example, CNAPS chips contained a 64 SIMD array of 16-bit by 8-bit multipliers, and several CNAPS chips could be connected together with a sequencer (Hammerstrom, 1990). The Synapse-1 system was based on a custom systolic multiply-accumulate chip called the MA-16, which performed sixteen 16-bit multiplications at a time (Ramacher et al., 1991). The system concatenated MA-16 chips and had custom hardware to do activation functions.

Twenty-five SPERT-II workstations, accelerated by the T0 custom ASIC, were deployed starting in 1995 to do both NN training and inference for speech recognition (Asanović et al., 1998). The 40-MHz T0 added vector instructions to the MIPS instruction set architecture. The eight-lane vector unit could produce up to sixteen 32-bit arithmetic results per clock cycle based on 8-bit and 16-bit inputs, making it 25 times faster at inference and 20 times faster at training than a SPARC-20 workstation. They found that 16 bits were insufficient for training, so they used two 16-bit words instead, which doubled training time. To overcome that drawback, they introduced "bunches" (batches) of 32–1000 data sets to reduce time spent updating weights, which made it faster than training with one word but no batches.

We use the phrase *Image Processing Unit* for Paintbox to identify this emerging class of processor, but this is not the first use of the term. The earliest use we can find is 1999, when the Sony Playstation put the name on a chip that was basically an MPEG2 decoder (Sony/Toshiba, 1999). In 2006, Freescale used IPU to name part of the i.MX31 Applications Processor, which is closer to the more generic way we interpret it (Freescale as part of i.MX31 Applications Processor, 2006).

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., 2016. Tensor-flow: large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

Adolf, R., Rama, S., Reagen, B., Wei, G.Y., Brooks, D., 2016. Fathom: reference workloads for modern deep learning methods. In: IEEE International Symposium on Workload Characterization (IISWC).

Amodei, D., et al., 2015. Deep speech 2: end-to-end speech recognition in English and mandarin, arXiv:1512.02595.

Asanović, K., 2002. Programmable neurocomputing. In: Arbib, M.A. (Ed.), The Handbook of Brain Theory and Neural Networks, second ed. MIT Press, Cambridge, MA. ISBN: 0-262-01197-2. https://people.eecs.berkeley.edu/∼krste/papers/neurocomputing.pdf.

Asanović, K., Beck, A., Johnson, J., Wawrzynek, J., Kingsbury, B., Morgan, N., 1998. Training neural networks with Spert-II. In: Sundararajan, N., Saratchandran, P. (Eds.), Parallel Architectures for Artificial Networks: Paradigms and Implementations. IEEE Computer Society Press. ISBN: 0-8186-8399-6. (Chapter 11) https://people.eecs.berkeley.edu/∼krste/papers/annbook.pdf.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K., 2012. Chisel: constructing hardware in a Scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference, pp. 1216–1225.

Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R., 1968. The ILLIAC IV computer. IEEE Trans. Comput. 100 (8), 746–757.

Bhattacharya, S., Lane, N.D., 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM, pp. 176–189.

Brunhaver, J., 2014. PhD thesis. Stanford.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D., Anderson, J.H., 2013. LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Trans. Embed. Comput. Syst. 13 (2).

Canny, J., et al., 2015. Machine learning at the limit. In: IEEE International Conference on Big Data.

Caulfield, A.M., Chung, E.S., Putnam, A., Haselman, H.A.J.F.M., Humphrey, S.H.M., Daniel, P.K.J.Y.K., Ovtcharov, L.T.M.K., Lanka, M.P.L.W.S., Burger, D.C.D., 2016. A cloud-scale acceleration architecture. In: MICRO Conference.

Charlesworth, A.E., 1981. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. Computer 9, 18–27.

Clark, J., October 26, 2015. Google Turning Its Lucrative Web Search Over to AI Machines. Bloomberg Technology, www.bloomberg.com.

Dally, W.J., 2002. Computer architecture is all about interconnect. In: Proceedings of the 8th International Symposium High Performance Computer Architecture.

Freescale as part of i.MX31 Applications Processor, 2006. http://cache.freescale.com/files/32bit/doc/white_paper/IMX31MULTIWP.pdf.

Galal, S., Shacham, O., Brunhaver II, J.S., Pu, J., Vassiliev, A., Horowitz, M., 2013. FPU generator for design space exploration. In: 21st IEEE Symposium on Computer Arithmetic (ARITH).

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M., 2010. Understanding sources of inefficiency in general-purpose chips. ACM SIGARCH Comput. Architect. News 38 (3), 37–47.

Hammerstrom, D., 1990. A VLSI architecture for high-performance, low-cost, on-chip learning. In: IJCNN International Joint Conference on Neural Networks.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Identity mappings in deep residual networks. Also in arXiv preprint arXiv:1603.05027.

Huang, M., Wu, D., Yu, C.H., Fang, Z., Interlandi, M., Condie, T., Cong, J., 2016. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, pp. 456–469.

Iandola, F., 2016. Exploring the Design Space of Deep Convolutional Neural Networks at Large Scale (Ph.D. dissertation). UC Berkeley.

Ienne, P., Cornu, T., Kuhn, G., 1996. Special-purpose digital hardware for neural networks: an architectural survey. J. VLSI Signal Process. Syst. Signal Image Video Technol. 13 (1).

Jouppi, N., 2016. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com.

Jouppi, N., Young, C., Patil, N., Patterson, D., Agrawal, G., et al., 2017. Datacenter performance analysis of a matrix processing unit. In: 44th International Symposium on Computer Architecture.

Karpathy, A., et al., 2014. Large-scale video classification with convolutional neural networks. CVPR.

Krizhevsky, A., Sutskever, I., Hinton, G., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst.

Kung, H.T., Leiserson, C.E., 1980. Algorithms for VLSI processor arrays. Introduction to VLSI systems.

Lee, Y., Waterman, A., Cook, H., Zimmer, B., Keller, B., Puggelli, A., Kwak, J., Jevtic, R., Bailey, S., Blagojevic, M., Chiu, P.-F., Avizienis, R., Richards, B., Bachrach, J., Patterson, D., Alon, E., Nikolic, B., Asanovic, K., 2016. An agile approach to building RISC-V microprocessors. IEEE Micro 36 (2), 8–20.

Lewis-Kraus, G., 2016. The Great A.I. Awakening. New York Times Magazine.

Nielsen, M., 2016. Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/.

Nvidia, 2016. Tesla GPU Accelerators For Servers. http://www.nvidia.com/object/teslaservers.html.

Olofsson, A., 2011. Debunking the myth of the $100 M ASIC. EE Times. http://www.eetimes.com/author.asp?section_id=36&doc_id=1266014.

Ovtcharov, K., Ruwase, O., Kim, J.Y., Fowers, J., Strauss, K., Chung, E.S., 2015a. Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper. https://www.microsoft.com/en-us/research/publication/accelerating-deepconvolutional-neural-networks-using-specialized-hardware/.

Ovtcharov, K., Ruwase, O., Kim, J.Y., Fowers, J., Strauss, K., Chung, E.S., 2015b. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In: 2015 IEEE Hot Chips 27 Symposium.

Patterson, D., Nikolić, B., 7/25/2015, Agile Design for Hardware, Parts I, II, and III. EE Times, http://www.eetimes.com/author.asp?doc_id=1327239.

Patterson, D.A., Ditzel, D.R., 1980. The case for the reduced instruction set computer. ACM SIGARCH Comput. Architect. News 8 (6), 25–33.

Prabhakar, R., Koeplinger, D., Brown, K.J., Lee, H., De Sa, C., Kozyrakis, C., Olukotun, K., 2016. Generating configurable hardware from parallel patterns. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, pp. 651–665.

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In: 41st International Symposium on Computer Architecture.

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2015. A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro. 35 (3).

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2016. A reconfigurable fabric for accelerating large-scale datacenter services. Commun. ACM.

Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C., Horowitz, M.A., 2015. Convolution engine: balancing efficiency & flexibility in specialized computing. Commun. ACM 58 (4).

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S., 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Not. 48 (6), 519–530.

Ramacher, U., Beichter, J., Raab, W., Anlauf, J., Bruels, N., Hachmann, A., Wesseling, M., 1991. Design of a 1st generation neurocomputer. VLSI Design of Neural Networks. Springer, USA.

Rau, B.R., Fisher, J.A., 1993. Instruction-level parallelism. J. Supercomput. 235, Springer Science & Business Media.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., 2015. Imagenet large scale visual recognition challenge. Int. J. Comput. Vis. 115 (3).

Sergio Guadarrama, 2015. BVLC googlenet. https://github.com/BVLC/caffe/tree/master/models/bvlc_googlenet.

Shao, Y.S., Brooks, D., 2015. Research infrastructures for hardware accelerators. Synth. Lect. Comput. Architect. 10 (4), 1–99.

Sharangpani, H., Arora, K., 2000. Itanium processor microarchitecture. IEEE Micro 20 (5), 24–43.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. Nature 529 (7587).

Smith, J.E., 1982. Decoupled access/execute computer architectures. In: Proceedings of the 11th International Symposium on Computer Architecture.

Sony/Toshiba, 1999. 'Emotion Engine' in PS2 ("IPU is basically an MPEG2 decoder…"). http://www.cpu-collection.de/?l0=co&l1=Sony&l2=Emotion+Engine, http://arstechnica.com/gadgets/2000/02/ee/3/.

Steinberg, D., 2015. Full-Chip Simulations, Keys to Success. In: Proceedings of the Synopsys Users Group (SNUG) Silicon Valley 2015.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2015. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

TensorFlow Tutorials, 2016. https://www.tensorflow.org/versions/r0.12/tutorials/index.html.

Tung, L., 2016. Google Translate: 'This landmark update is our biggest single leap in 10 years', ZDNet. http://www.zdnet.com/article/google-translate-this-landmarkupdate-is-our-biggest-single-leap-in-10years/.

Vanhoucke, V., Senior, A., Mao, M.Z., 2011. Improving the speed of neural networks on CPUs. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37631.pdf.

Wu, Y., Schuster, M., Chen, Z., Le, Q., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., Dean, J., 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. http://arxiv.org/abs/1609.08144.

M.10 The History of Magnetic Storage, RAID, and I/O Buses (Appendix D)

Mass storage is a term used there to imply a unit capacity in excess of one million alphanumeric characters …

Hoagland [1963]

The variety of storage I/O and issues leads to a varied history for the rest of the story. (Smotherman [1989] explored the history of I/O in more depth.) This section discusses magnetic storage, RAID, and I/O buses and controllers. Jain [1991] and Lazowska et al. [1984] are books for those interested in learning more about queuing theory.

Magnetic Storage

Magnetic recording was invented to record sound, and by 1941 magnetic tape was able to compete with other storage devices. It was the success of the ENIAC in 1947 that led to the push to use tapes to record digital information. Reels of magnetic tapes dominated removable storage through the 1970s. In the 1980s, the IBM 3480 cartridge became the *de facto* standard, at least for mainframes. It can transfer at 3 MB/sec by reading 18 tracks in parallel. The capacity is just 200 MB for this 1/2-inch tape. The 9840 cartridge, used by StorageTek in the Powder-Horn, transfers at 10 MB/sec and stores 20,000 MB. This device records the tracks in a zigzag fashion rather than just longitudinally, so that the head reverses direction to follow the track. This technique is called *serpentine recording*. Another 1/2-inch tape is Digital Linear Tape; the DLT7000 stores 35,000 MB and transfers at 5 MB/sec. Its competitor is helical scan, which rotates the head to get the increased recording density. In 2001, the 8-mm helical-scan tapes contain 20,000 MB and transfer at about 3 MB/sec. Whatever their density and cost, the serial nature of tapes creates an appetite for storage devices with random access.

In 1953, Reynold B. Johnson of IBM picked a staff of 15 scientists with the goal of building a radically faster random access storage system than tape. The goal was to have the storage equivalent of 50,000 standard IBM punch cards and to fetch the data in a single second. Johnson's disk drive design was simple but untried: The magnetic read/write sensors would have to float a few thousandths of an inch above the continuously rotating disk. Twenty-four months later the team emerged with the functional prototype. It weighed 1 ton and occupied about 300 cubic feet of space. The RAMAC-350 (Random Access Method of Accounting Control) used 50 platters that were 24 inches in diameter, rotated at 1200 RPM, with a total capacity of 5 MB and an access time of 1 second.

Starting with the RAMAC, IBM maintained its leadership in the disk industry, with its storage headquarters in San Jose, California, where Johnson's team did its work. Many of the future leaders of competing disk manufacturers started their careers at IBM, and many disk companies are located near San Jose.

Although RAMAC contained the first disk, a major breakthrough in magnetic recording was found in later disks with air-bearing read/write heads, where the head would ride on a cushion of air created by the fast-moving disk surface. This cushion meant the head could both follow imperfections in the surface and yet be very close to the surface. Subsequent advances have come largely from improved quality of components and higher precision. In 2001, heads flew 2 to 3 microinches above the surface, whereas in the RAMAC drive they were 1000 microinches away.

Moving-head disks quickly became the dominant high-speed magnetic storage, although their high cost meant that magnetic tape continued to be used extensively until the 1970s. The next important development for hard disks was the removable hard disk drive developed by IBM in 1962; this made it possible to share the expensive drive electronics and helped disks overtake tapes as the preferred storage medium. The IBM 1311 disk in 1962 had an areal density of 50,000 bits per square inch and a cost of about $800 per megabyte. IBM also invented the floppy disk drive in 1970, originally to hold microcode for the IBM 370 series. Floppy disks became popular with the PC about 10 years later.

The second major disk breakthrough was the so-called Winchester disk design in about 1973. Winchester disks benefited from two related properties. First, integrated circuits lowered the costs of not only CPUs but also of disk controllers and the electronics to control disk arms. Reductions in the cost of the disk electronics made it unnecessary to share the electronics and thus made nonremovable disks economical. Since the disk was fixed and could be in a sealed enclosure, both the environmental and control problems were greatly reduced. Sealing the system allowed the heads to fly closer to the surface, which in turn enabled increases in areal density. The first sealed disk that IBM shipped had two spindles, each with a 30 MB disk; the moniker "30-30" for the disk led to the name Winchester. (America's most popular sporting rifle, the Winchester 94, was nicknamed the "30-30" after the caliber of its cartridge.) Winchester disks grew rapidly in popularity in the 1980s, completely replacing removable disks by the middle of that decade. Before this time, the cost of the electronics to control the disk meant that the media had to be removable.

As mentioned in Appendix D, as DRAMs started to close the areal density gap and appeared to be catching up with disk storage, internal meetings at IBM called into question the future of disk drives. Disk designers concluded that disks must improve at 60% per year to forestall the DRAM threat, in contrast to the historical 29% per year. The essential enabler was magnetoresistive heads, with giant magnetoresistive heads enabling the current densities. Because of this competition, the gap in time between when a density record is achieved in the lab and when a disk is shipped with that density has closed considerably.

The personal computer created a market for small form factor (SFF) disk drives, since the 14-inch disk drives used in mainframes were bigger than the PC. In 2006, the 3.5-inch drive was the market leader, although the smaller 2.5-inch drive required for laptop computers was significant in sales volume. It remains to be seen whether handheld devices such as iPODs or video cameras, which require even smaller disks, will remain significant in sales volume. For example, 1.8-inch drives were developed in the early 1990s for palmtop computers, but that market chose Flash instead and 1.8-inch drives disappeared.

RAID

The SFF hard disks for PCs in the 1980s led a group at Berkeley to propose redundant arrays of inexpensive disks (RAID). This group had worked on the reduced instruction set computer effort and so expected much faster CPUs to become available. They asked: What could be done with the small disks that accompanied their PCs? and What could be done in the area of I/O to keep up with much faster processors? They argued to replace one mainframe drive with 50 small drives to gain much greater performance from that many independent arms. The many small drives even offered savings in power consumption and floor space. The downside of many disks was much lower mean time to failure (MTTF). Hence, on their own they reasoned out the advantages of redundant disks and rotating parity to address how to get greater performance with many small drives yet have reliability as high as that of a single mainframe disk.

The problem they experienced when explaining their ideas was that some researchers had heard of disk arrays with some form of redundancy, and they didn't understand the Berkeley proposal. Hence, the first RAID paper [Patterson, Gibson, and Katz 1987] is not only a case for arrays of SFF disk drives but also something of a tutorial and classification of existing work on disk arrays. Mirroring (RAID 1) had long been used in fault-tolerant computers such as those sold by Tandem. Thinking Machines had arrays with 32 data disks and 7 check disks using ECC for correction (RAID 2) in 1987, and Honeywell Bull had a RAID 2 product even earlier. Also, disk arrays with a single parity disk had been used in scientific computers in the same time frame (RAID 3). Their paper then described a single parity disk with support for sector accesses (RAID 4) and rotated parity (RAID 5). Chen et al. [1994] surveyed the original RAID ideas, commercial products, and more recent developments.

Unknown to the Berkeley group, engineers at IBM working on the AS/400 computer also came up with rotated parity to give greater reliability for a collection of large disks. IBM filed a patent on RAID 5 before the Berkeley group wrote their paper. Patents for RAID 1, RAID 2, and RAID 3 from several companies predate the IBM RAID 5 patent, which has led to plenty of courtroom action.

The Berkeley paper was written before the World Wide Web, but it captured the imagination of many engineers, as copies were faxed around the world. One engineer at what is now Seagate received seven copies of the paper from friends and customers. EMC had been a supplier of DRAM boards for IBM computers, but around 1988 new policies from IBM made it nearly impossible for EMC to continue to sell IBM memory boards. Apparently, the Berkeley paper also crossed the desks of EMC executives, and they decided to go after the market dominated by IBM disk storage products instead. As the paper advocated, their model was to use many small drives to compete with mainframe drives, and EMC announced a RAID product in 1990. It relied on mirroring (RAID 1) for reliability; RAID 5 products came much later for EMC. Over the next year, Micropolis offered a RAID 3 product, Compaq offered a RAID 4 product, and Data General, IBM, and NCR offered RAID 5 products.

The RAID ideas soon spread to the rest of the workstation and server industry. An article explaining RAID in *Byte* magazine (see Anderson [1990]) led to RAID products being offered on desktop PCs, which was something of a surprise to the Berkeley group. They had focused on performance with good availability, but higher availability was attractive to the PC market.

Another surprise was the cost of the disk arrays. With redundant power supplies and fans, the ability to "hot swap" a disk drive, the RAID hardware controller itself, the redundant disks, and so on, the first disk arrays cost many times the cost of the disks. Perhaps as a result, the "inexpensive" in RAID morphed into "independent." Many marketing departments and technical writers today know of RAID only as "redundant arrays of independent disks."

The EMC transformation was successful; in 2006, EMC was the leading supplier of storage systems, and NetApp was the leading supplier of Network-Attached Storage systems. RAID was a $30 billion industry in 2006, and more than 80% of the non-PC drive sales were found in RAIDs. In recognition of their role, in 1999 Garth Gibson, Randy Katz, and David Patterson received the IEEE Reynold B. Johnson Information Storage Award "for the development of Redundant Arrays of Inexpensive Disks (RAID)."

I/O Buses and Controllers

The ubiquitous microprocessor inspired not only the personal computers of the 1970s but also the trend in the late 1980s and 1990s of moving controller functions into I/O devices. I/O devices have continued this trend by moving controllers into the devices themselves. These devices are called *intelligent devices*, and some bus standards (e.g., SCSI) have been created specifically for them. Intelligent devices can relax the timing constraints by handling many low-level tasks themselves and queuing the results. For example, many SCSI-compatible disk drives include a track buffer on the disk itself, supporting read ahead and connect/disconnect. Thus, on a SCSI string some disks can be seeking and others loading their track buffer while one is transferring data from its buffer over the SCSI bus. The controller in the original RAMAC, built from vacuum tubes, only needed to move the head over the desired track, wait for the data to pass under the head, and transfer data with calculated parity. SCSI, which stands for *small computer systems interface*, is an example of one company inventing a bus and generously encouraging other companies to build devices that would plug into it. Shugart created this bus, originally called SASI. It was later standardized by the IEEE.

There have been several candidates to be the successor to SCSI, with the current leading contender being Fibre Channel Arbitrated Loop (FC-AL). The SCSI committee continues to increase the clock rate of the bus, giving this standard a new life, and SCSI is lasting much longer than some of its proposed successors. With the creation of serial interfaces for SCSI ("Serial Attach SCSI") and ATA ("Serial ATA"), they may have very long lives.

Perhaps the first multivendor bus was the PDP-11 Unibus in 1970 from DEC. Alas, this open-door policy on buses is in contrast to companies with proprietary buses using patented interfaces, thereby preventing competition from plug-compatible vendors. Making a bus proprietary also raises costs and lowers the number of available I/O devices that plug into it, since such devices must have an interface designed just for that bus. The PCI bus pushed by Intel represented a return to open, standard I/O buses inside computers. Its immediate successor is PCI-X, with Infiniband under development in 2000. Both were standardized by multicompany trade associations.

The machines of the RAMAC era gave us I/O interrupts as well as storage devices. The first machine to extend interrupts from detecting arithmetic abnormalities to detecting asynchronous I/O events is credited as the NBS DYSEAC in 1954 [Leiner and Alexander 1954]. The following year, the first machine with DMA was operational, the IBM SAGE. Just as today's DMA has, the SAGE had address counters that performed block transfers in parallel with CPU operations.

The early IBM 360s pioneered many of the ideas that we use in I/O systems today. The 360 was the first commercial machine to make heavy use of DMA, and it introduced the notion of I/O programs that could be interpreted by the device. Chaining of I/O programs was an important feature. The concept of channels introduced in the 360 corresponds to the I/O bus of today.

Myer and Sutherland [1968] wrote a classic paper on the trade-off of complexity and performance in I/O controllers. Borrowing the religious concept of the "wheel of reincarnation," they eventually noticed they were caught in a loop of continuously increasing the power of an I/O processor until it needed its own simpler coprocessor. Their quote in Appendix D captures their cautionary tale.

The IBM mainframe I/O channels, with their I/O processors, can be thought of as an inspiration for Infiniband, with their processors on their Host Channel Adaptor cards.

References

Anderson, D. [2003]. "You don't know jack about disks," *Queue* 1:4 (June), 20–30.

Anderson, D., J. Dykes, and E. Riedel [2003]. "SCSI vs. ATA—more than an interface," *Proc. 2nd USENIX Conf. on File and Storage Technology (FAST '03)*, March 31–April 2, 2003, San Francisco.

Anderson, M. H. [1990]. "Strength (and safety) in numbers (RAID, disk storage technology)," *Byte* 15:13 (December), 337–339.

Anon. et al. [1985]. *A Measure of Transaction Processing Power*, Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation*, 31:7 (April), 112–118.

Bashe, C. J., W. Buchholz, G. V. Hawkins, J. L. Ingram, and N. Rochester [1981]. "The architecture of IBM's early computers," *IBM J. Research and Development* 25:5 (September), 363–375.

Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh [1986]. *IBM's Early Computers*, MIT Press, Cambridge, Mass.

Blaum, M., J. Brady, J. Bruck, and J. Menon [1994]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures," *Proc. 21st Annual Int'l. Symposium on Computer Architecture (ISCA)*, April 18–21, 1994, Chicago, 245–254.

Blaum, M., J. Brady, J. Bruck, and J. Menon [1995]. "EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. on Computers* 44:2 (February), 192–202.

Blaum, M., J. Brady, J., Bruck, J. Menon, and A. Vardy [2001]. "The EVENODD code and its generalization," in H. Jin, T. Cortes, and R. Buyya, eds., *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, IEEE & Wiley Press, New York, 187–208.

Blaum, M., J. Bruck, and A. Vardy [1996]. "MDS array codes with independent parity symbols," *IEEE Trans. on Information Theory*, IT-42 (March), 529–542.

Brady, J. T. [1986]. "A theory of productivity in the creative process," *IEEE CG&A* (May), 25–34.

Brown, A., and D. A. Patterson [2000]. "Towards maintainability, availability, and growth benchmarks: A case study of software RAID systems." *Proc. 2000 USENIX Annual Technical Conf.*, June 18–23, San Diego, Calif.

Bucher, I. V., and A. H. Hayes [1980]. "I/O performance measurement on Cray-1 and CDC 7000 computers," *Proc. Computer Performance Evaluation Users Group, 16th Meeting*, October 20–23, 1980, Orlando, Fl., 245–254.

Chen, P. M., G. A. Gibson, R. H. Katz, and D. A. Patterson [1990]. "An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 22–25, 1990, Boulder, Colo.

Chen, P. M., and E. K. Lee [1995]. "Striping in a RAID level 5 disk array," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 15–19, 1995, Ottawa, Canada, 136–145.

Chen, P. M., E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson [1994]. "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys* 26:2 (June), 145–188.

Corbett, P., B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar [2004]. "Row-diagonal parity for double disk failure correction," *Proc. 3rd USENIX Conf. on File and Storage Technology (FAST '04)*, March 31–April 2, 2004, San Francisco.

Denehy, T. E., J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau [2004]. "Deconstructing storage arrays," *Proc. 11th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 7–13, 2004, Boston, Mass., 59–71.

Doherty, W. J., and R. P. Kelisky [1979]. "Managing VM/CMS systems for user effectiveness," *IBM Systems J.* 18:1, 143–166.

Douceur, J. R., and W. J. Bolosky [1999]. "A large scale study of file-system contents," *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, May 1–9, 1999, Atlanta, Ga., 59–69.

Enriquez, P. [2001]. "What happened to my dial tone? A study of FCC service disruption reports," poster, *Richard Tapia Symposium on the Celebration of Diversity in Computing*, October 18–20, 2001, Houston, Tex.

Friesenborg, S. E., and R. J. Wicks [1985]. *DASD Expectations: The 3380, 3380-23, and MVS/XA*, Tech. Bulletin GG22-9363-02, IBM Washington Systems Center, Gaithersburg, Md.

Gibson, G. A. [1992]. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*, ACM Distinguished Dissertation Series, MIT Press, Cambridge, Mass.

Goldstein, S. [1987]. *Storage Performance—An Eight Year Outlook*, Tech. Rep. TR 03.308-1, IBM Santa Teresa Laboratory, San Jose, Calif.

Gray, J. [1990]. "A census of Tandem system availability between 1985 and 1990," *IEEE Trans. on Reliability*, 39:4 (October), 409–418.

Gray, J. (ed.) [1993]. *The Benchmark Handbook for Database and Transaction Processing Systems*, 2nd ed., Morgan Kaufmann, San Francisco.

Gray, J., and A. Reuter [1993]. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco.

Gray, J., and D. P. Siewiorek [1991]. "High-availability computer systems." *Computer* 24:9 (September), 39–48.

Gray, J., and C. van Ingen [2005]. *Empirical Measurements of Disk Failure Rates and Error Rates*," MSR-TR-2005-166, Microsoft Research, Redmond, Wash.

Gurumurthi, S., A. Sivasubramaniam, and V. Natarajan [2005]. Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management, *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June, 38–49.

Henly, M., and B. McNutt [1989]. *DASD I/O Characteristics: A Comparison of MVS to VM*, Tech. Rep. TR 02.1550, IBM General Products Division, San Jose, Calif.

Hewlett-Packard. [1998]. "HP's '5NINES:5MINUTES' vision extends leadership and re-defines high availability in mission-critical environments," February 10, *www.future.enterprisecomputing.hp.com/ia64/news/5nines_vision_pr.html*.

Hoagland, A. S. [1963]. *Digital Magnetic Recording*, Wiley, New York.

Hospodor, A. D., and A. S. Hoagland [1993]. "The changing nature of disk controllers." *Proc. IEEE* 81:4 (April), 586–594.

IBM. [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0, IBM, White Plains, N.Y., 11–82.

Imprimis. [1989]. *Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB*, Document No. 64402302, Imprimis, Dallas, Tex.

Jain, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley, New York.

Katz, R. H., D. A. Patterson, and G. A. Gibson [1989]. "Disk system architectures for high performance computing," *Proc. IEEE* 77:12 (December), 1842–1858.

Kim, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November), 978–988.

Kuhn, D. R. [1997]. "Sources of failure in the public switched telephone network," *IEEE Computer* 30:4 (April), 31–36.

Lambright, D. [2000]. "Experiences in measuring the reliability of a cache-based storage system," *Proc. of First Workshop on Industrial Experiences with Systems Software (WIESS 2000), Co-Located with the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, October 22, 2000, San Diego, Calif.

Laprie, J.-C. [1985]. "Dependable computing and fault tolerance: Concepts and terminology," *Proc. 15th Annual Int'l. Symposium on Fault-Tolerant Computing*, June 19–21, 1985, Ann Arbor, Mich., 2–11.

Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik [1984]. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*, Prentice Hall, Englewood Cliffs, N.J. (Although out of print, it is available online at *www.cs.washington.edu/homes/lazowska/qsp/.)*

Leiner, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April), 57–81.

Leiner, A. L., and S. N. Alexander [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March), 1–10.

Maberly, N. C. [1966]. *Mastering Speed Reading*, New American Library, New York.

Major, J. B. [1989]. "Are queuing models within the grasp of the unwashed?" *Proc. Int'l. Conf. on Management and Performance Evaluation of Computer Systems*, December 11–15, 1989, Reno, Nev., 831–839.

Mueller, M., L. C. Alves, W. Fischer, M. L. Fair, and I. Modi [1999]. "RAS strategy for IBM S/390 G5 and G6," *IBM J. Research and Development*, 43:5–6 (September–November), 875–888.

Murphy, B., and T. Gent [1995]. "Measuring system and software reliability using an automated data collection process," *Quality and Reliability Engineering International*, 11:5 (September–October), 341–353.

Myer, T. H., and I. E. Sutherland [1968]. "On the design of display processors," *Communications of the ACM*, 11:6 (June), 410–414.

National Storage Industry Consortium. [1998]. "Tape Roadmap," *www.nsic.org*.

Nelson, V. P. [1990]. "Fault-tolerant computing: Fundamental concepts," *Computer* 23:7 (July), 19–25.

Nyberg, C. R., T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet [1994]. "AlphaSort: A RISC machine sort," *Proc. ACM SIGMOD*, May 24–27, 1994, Minneapolis, Minn.

Okada, S., S. Okada, Y. Matsuda, T. Yamada, and A. Kobayashi [1999]. "System on a chip for digital still camera," *IEEE Trans. on Consumer Electronics* 45:3 (August), 584–590.

Patterson, D. A., G. A. Gibson, and R. H. Katz [1987]. *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Tech. Rep. UCB/CSD 87/391, University of California, Berkeley. Also appeared in *Proc. ACM SIGMOD*, June 1–3, 1988, Chicago, 109–116.

Pavan, P., R. Bez, P. Olivo, and E. Zanoni [1997]. "Flash memory cells—an overview," *Proc. IEEE* 85:8 (August), 1248–1271.

Robinson, B., and L. Blount [1986]. *The VM/HPO 3880-23 Performance Results*, IBM Tech. Bulletin GG66-0247-00, IBM Washington Systems Center, Gaithersburg, Md.

Salem, K., and H. Garcia-Molina [1986]. "Disk striping," *Proc. 2nd Int'l. IEEE Conf. on Data Engineering*, February 5–7, 1986, Washington, D.C., 249–259.

Scranton, R. A., D. A. Thompson, and D. W. Hunter [1983]. *The Access Time Myth*, Tech. Rep. RC 10197 (45223), IBM, Yorktown Heights, N.Y.

Seagate. [2000]. *Seagate Cheetah 73 Family: ST173404LW/LWV/LC/LCV Product Manual*, Vol. 1, Seagate, Scotts Valley, Calif. (*www.seagate.com/support/disc/manuals/scsi/29478b.pdf*).

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15. Reprinted in *Computer Architecture Readings*, M. D. Hill, N. P. Jouppi, and G. S. Sohi, eds., Morgan Kaufmann, San Francisco, 1999, 451–461.

Talagala, N. [2000]. "Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks," Ph.D. dissertation, Computer Science Division, University of California, Berkeley.

Talagala, N., and D. Patterson [1999]. *An Analysis of Error Behavior in a Large Storage System*, Tech. Report UCB//CSD-99-1042, Computer Science Division, University of California, Berkeley.

Talagala, N., R. Arpaci-Dusseau, and D. Patterson [2000]. *Micro-Benchmark Based Extraction of Local and Global Disk Characteristics*, CSD-99-1063, Computer Science Division, University of California, Berkeley.

Talagala, N., S. Asami, D. Patterson, R. Futernick, and D. Hart [2000]. "The art of massive storage: A case study of a Web image archive," *IEEE Computer* (November), 22–28.

Thadhani, A. J. [1981]. "Interactive user productivity," *IBM Systems J.* 20:4, 407–423.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., 2016. TensorFlow: A System for Large-Scale Machine Learning. In: OSDI (November), vol. 16, pp. 265–283.

Adolf, R., Rama, S., Reagen, B., Wei, G.Y., Brooks, D., 2016. Fathom: reference workloads for modern deep learning methods. In: IEEE International Symposium on Workload Characterization (IISWC).

Adve, S.V., Gharachorloo, K., 1996. Shared memory consistency models: a tutorial. IEEE Comput. 29 (12), 66–76.

Adve, S.V., Hill, M.D., 1990. Weak ordering: a new definition. In: Proceedings of 17th Annual International Symposium on Computer Architecture (ISCA), May 28–31, 1990, Seattle, Washington, pp. 2–14.

Agarwal, A., 1987. Analysis of Cache Performance for Operating Systems and Multiprogramming (Ph.D. thesis). Tech. Rep. No. CSL-TR-87-332. Stanford University, Palo Alto, CA.

Agarwal, A., 1991. Limits on interconnection network performance. IEEE Trans. Parallel Distrib. Syst. 2 (4), 398–412.

Agarwal, A., Pudar, S.D., 1993. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In: 20th Annual International Symposium on Computer Architecture (ISCA), May 16–19, 1993, San Diego, California. Also appears in Computer Architecture News 21:2 (May), 179–190, 1993.

Agarwal, A., Hennessy, J.L., Simoni, R., Horowitz, M.A., 1988. An evaluation of directory schemes for cache coherence. In: Proceedings of 15th International Symposium on Computer Architecture (June), pp. 280–289.

Agarwal, A., Kubiatowicz, J., Kranz, D., Lim, B.-H., Yeung, D., D'Souza, G., Parkin, M., 1993. Sparcle: an evolutionary processor design for large-scale multiprocessors. IEEE Micro 13, 48–61.

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K., Kranz, D., 1995. The MIT Alewife machine: architecture and performance. In: International Symposium on Computer Architecture (Denver, CO), June, 2–13.

Agerwala, T., Cocke, J., 1987. High Performance Reduced Instruction Set Processors. IBM Tech. Rep. RC12434, IBM, Armonk, NY.

Akeley, K., Jermoluk, T., 1988. High-performance polygon rendering. In: Proceedings of 15th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1988), August 1–5, 1988, Atlanta, GA, pp. 239–246.

Alexander, W.G., Wortman, D.B., 1975. Static and dynamic characteristics of XPL programs. IEEE Comput. 8 (11), 41–46.

Alles, A., 1995. ATM Internetworking. White Paper (May). Cisco Systems, Inc., San Jose, CA. www.cisco.com/warp/public/614/12.html.

Alliant, 1987. Alliant FX/Series: Product Summary. Alliant Computer Systems Corp, Acton, MA.

Almasi, G.S., Gottlieb, A., 1989. Highly Parallel Computing. Benjamin/Cummings, Redwood City, CA.

Alverson, G., Alverson, R., Callahan, D., Koblenz, B., Porterfield, A., Smith, B., 1992. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 16–20, 1992, Minneapolis, MN, pp. 188–197.

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of AFIPS Spring Joint Computer Conference, April 18–20, 1967, Atlantic City, NJ, pp. 483–485.

Amdahl, G.M., Blaauw, G.A., Brooks Jr., F.P., 1964. Architecture of the IBM System 360. IBM J. Res. Dev. 8 (2), 87–101.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., Chen, J., 2016. Deep speech 2: End-to-end speech recognition in english and mandarin. In: International Conference on Machine Learning (June), pp. 173–182.

Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W., 1996. Treadmarks: shared memory computing on networks of workstations. IEEE Comput. 29 (2), 18–28.

Anderson, M.H., 1990. Strength (and safety) in numbers (RAID, disk storage technology). Byte 15 (13), 337–339.

Anderson, D., 2003. You don't know jack about disks. Queue 1 (4), 20–30.

Anderson, D.W., Sparacio, F.J., Tomasulo, R.M., 1967. The IBM 360 Model 91: processor philosophy and instruction handling. IBM J. Res. Dev. 11 (1), 8–24.

Anderson, T.E., Culler, D.E., Patterson, D., 1995. A case for NOW (networks of workstations). IEEE Micro 15 (1), 54–64.

Anderson, D., Dykes, J., Riedel, E., 2003. SCSI vs. ATA—more than an interface. In: Proceedings of 2nd USENIX Conference on File and Storage Technology (FAST'03), March 31–April 2.

Ang, B., Chiou, D., Rosenband, D., Ehrlich, M., Rudolph, L., Arvind, A., 1998. StarT-Voyager: a flexible platform for exploring scalable SMP issues. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 7–13, 1998, Orlando, FL.

Anjan, K.V., Pinkston, T.M., 1995. An efficient, fully-adaptive deadlock recovery scheme: Disha. In: Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA), June 22–24, 1995, Santa Margherita, Italy.

Anon. et al., 1985. A Measure of Transaction Processing Power. Tandem Tech. Rep. TR85.2. Also appears in *Datamation* 31:7 (April), 112–118, 1985.

Apache Hadoop, 2011. http://hadoop.apache.org.

Archibald, J., Baer, J.-L., 1986. Cache coherence protocols: evaluation using a multiprocessor simulation model. ACM Trans. Comput. Syst. 4 (4), 273–298.

Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M., 2009. Above the Clouds: A Berkeley View of Cloud Computing, Tech. Rep. UCB/EECS-2009-28, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M., 2010. A view of cloud computing. Commun. ACM. 53 (4), 50–58.

Arpaci, R.H., Culler, D.E., Krishnamurthy, A., Steinberg, S.G., Yelick, K., 1995. Empirical evaluation of the CRAY-T3D: a compiler perspective. In: 22nd Annual International Symposium on Computer Architecture (ISCA), June 22–24, 1995, Santa Margherita, Italy.

Asanovic, K., 1998. Vector Microprocessors (Ph.D. thesis). Computer Science Division, University of California, Berkeley.

Asanović, K., 2002. Programmable neurocomputing. In: Arbib, M.A. (Ed.), The Handbook of Brain Theory and Neural Networks, second ed. MIT Press, Cambridge, MA. ISBN: 0-262-01197-2. https://people.eecs.berkeley.edu/∼krste/papers/neurocomputing.pdf.

Asanović, K., Beck, A., Johnson, J., Wawrzynek, J., Kingsbury, B., Morgan, N., 1998. Training neural networks with Spert-II. In: Sundararajan, N., Saratchandran, P. (Eds.), Parallel Architectures for Artificial Networks: Paradigms and Implementations. IEEE Computer Society Press, California, USA. ISBN: 0-8186-8399-6. (Chapter 11) https://people.eecs.berkeley.edu/~krste/papers/annbook.pdf.

Associated Press, 2005. Gap Inc. shuts down two Internet stores for major overhaul. USATODAY.com, August 8, 2005.

Atanasoff, J.V., 1940. Computing Machine for the Solution of Large Systems of Linear Equations. Internal Report. Iowa State University, Ames.

Atkins, M., 1991. Performance and the i860 microprocessor. IEEE Micro 11 (5), 24–27. 72–78.

Austin, T.M., Sohi, G., 1992. Dynamic dependency analysis of ordinary programs. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 342–351.

Azizi, O., Mahesri, A., Lee, B.C., Patel, S.J., Horowitz, M., 2010. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In: Proceedings of the International Symposium on Computer Architecture, pp. 26–36.

Babbay, F., Mendelson, A., 1998. Using value prediction to increase the power of speculative execution hardware. ACM Trans. Comput. Syst. 16 (3), 234–270.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., Asanović, K., 2012. Chisel: constructing hardware in a Scala embedded language. In: Proceedings of the 49th Annual Design Automation Conference, pp. 1216–1225.

Baer, J.-L., Wang, W.-H., 1988. On the inclusion property for multi-level cache hierarchies. In: Proceedings of 15th Annual International Symposium on Computer Architecture, May 30–June 2, 1988, Honolulu, Hawaii, pp. 73–80.

Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K., 1991. The NAS parallel benchmarks. Int. J. Supercomput. Appl. 5, 63–73.

Bakoglu, H.B., Grohoski, G.F., Thatcher, L.E., Kaeli, J.A., Moore, C.R., Tattle, D.P., Male, W.E., Hardell, W.R., Hicks, D.A., Nguyen Phu, M., Montoye, R.K., Glover, W.T., Dhawan, S., 1989. IBM second-generation RISC processor organization. In: Proceedings of IEEE International Conference on Computer Design, September 30–October 4, 1989, Rye, NY, pp. 138–142.

Balakrishnan, H., Padmanabhan, V.N., Seshan, S., Katz, R.H., 1997. A comparison of mechanisms for improving TCP performance over wireless links. IEEE/ACM Trans. Netw. 5 (6), 756–769.

Ball, T., Larus, J., 1993. Branch prediction for free. In: Proceedings of ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), June 23–25, 1993, Albuquerque, NM, pp. 30 0–313.

Banerjee, U., 1979. Speedup of Ordinary Programs (Ph.D. thesis). Department of Computer Science, University of Illinois at Urbana-Champaign.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., 2003. Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, October 19–22, 2003, Bolton Landing, NY.

Barnes, G.H., Brown, R.M., Kato, M., Kuck, D.J., Slotnick, D.L., Stokes, R., 1968. The ILLIAC IV computer. IEEE Trans. Comput. 100 (8), 746–757.

Barroso, L.A., 2010. Warehouse scale computing [keynote address]. In: Proceedings of ACM SIGMOD, June 8–10, 2010, Indianapolis, IN.

Barroso, L.A., Hölzle, U., 2007. The case for energy-proportional computing. IEEE Comput. 40 (12), 33–37.

Barroso, L.A., Hölzle, U., 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan & Claypool, San Rafael, CA.

Barroso, L.A., Gharachorloo, K., Bugnion, E., 1998. Memory system characterization of commercial workloads. In: Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA), July 3–14, 1998, Barcelona, Spain, pp. 3–14.

Barroso, L.A., Clidaras, J., Hölzle, U., 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. Synth. Lect. Comput. Architect. 8 (3), 1–154.

Barroso, L.A., Marty, M., Patterson, D., Ranganathan, P., 2017. Attack of the killer microseconds. Commun. ACM 56(2).

Barton, R.S., 1961. A new approach to the functional design of a computer. In: Proceedings of Western Joint Computer Conference, May 9–11, 1961, Los Angeles, CA, pp. 393–396.

Bashe, C.J., Buchholz, W., Hawkins, G.V., Ingram, J.L., Rochester, N., 1981. The architecture of IBM's early computers. IBM J. Res. Dev. 25 (5), 363–375.

Bashe, C.J., Johnson, L.R., Palmer, J.H., Pugh, E.W., 1986. IBM's Early Computers. MIT Press, Cambridge, MA.

Baskett, F., Keller, T.W., 1977. An evaluation of the Cray-1 processor. In: Kuck, D.J., Lawrie, D.H., Sameh, A.H. (Eds.), High Speed Computer and Algorithm Organization. Academic Press, San Diego, pp. 71–84.

Baskett, F., Jermoluk, T., Solomon, D., 1988. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 10,000 lighted polygons per second. In: Proceedings of IEEE COMPCON, February 29–March 4, 1988, San Francisco, pp. 468–471.

BBN Laboratories, 1986. Butterfly Parallel Processor Overview, Tech. Rep. 6148. BBN Laboratories, Cambridge, MA.

Bell, C.G., 1984. The mini and micro industries. IEEE Comput. 17 (10), 14–30.

Bell, C.G., 1985. Multis: a new class of multiprocessor computers. Science 228 (6), 462–467.

Bell, C.G., 1989. The future of high performance computers in science and engineering. Commun. ACM 32 (9), 1091–1101.

Bell, G., Gray, J., 2001. Crays, Clusters and Centers, Tech. Rep. MSR-TR-2001-76. Microsoft Research, Redmond, WA.

Bell, C.G., Gray, J., 2002. What's next in high performance computing? CACM 45 (2), 91–95.

Bell, C.G., Newell, A., 1971. Computer Structures: Readings and Examples. McGraw-Hill, New York.

Bell, C.G., Strecker, W.D., 1976. Computer structures: what have we learned from the PDP-11? In: Third Annual International Symposium on Computer Architecture (ISCA), January 19–21, 1976, Tampa, FL, pp. 1–14.

Bell, C.G., Strecker, W.D., 1998. Computer structures: what have we learned from the PDP-11? In: 25 Years of the International Symposia on Computer Architecture (Selected Papers), ACM, New York, pp. 138–151.

Bell, C.G., Cady, R., McFarland, H., DeLagi, B., O'Laughlin, J., Noonan, R., Wulf, W., 1970. A new architecture for mini-computers: The DEC PDP-11. In: Proceedings of AFIPS Spring Joint Computer Conference, May 5–May 7, 1970, Atlantic City, NJ, pp. 657–675.

Bell, C.G., Mudge, J.C., McNamara, J.E., 1978. A DEC View of Computer Engineering. Digital Press, Bedford, MA.

Benes, V.E., 1962. Rearrangeable three stage connecting networks. Bell Syst. Tech. J. 41, 1481–1492.

Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., De Micheli, G., 2005. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. IEEE Trans. Parallel Distrib. Syst. 16 (2), 113–130.

Bhandarkar, D.P., 1995. Alpha Architecture and Implementations. Digital Press, Newton, MA.

Bhandarkar, D.P., Clark, D.W., 1991. Performance from architecture: comparing a RISC and a CISC with similar hardware organizations. In: Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 8–11, 1991, Palo Alto, CA, pp. 310–319.

Bhandarkar, D.P., Ding, J., 1997. Performance characterization of the Pentium Pro processor. In: Proceedings of Third International Symposium on High-Performance Computer Architecture, February 1–February 5, 1997, San Antonio, TX, pp. 288–297.

Bhattacharya, S., Lane, N.D., 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In: Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM, pp. 176–189.

Bhuyan, L.N., Agrawal, D.P., 1984. Generalized hypercube and hyperbus structures for a computer network. IEEE Trans. Comput. 32 (4), 322–333.

Bienia, C., Kumar, S., Jaswinder, P.S., Li, K., 2008. The Parsec Benchmark Suite: Characterization and Architectural Implications, Tech. Rep. TR-811-08. Princeton University, Princeton, NJ.

Bier, J., 1997. The evolution of DSP processors. In: Presentation at University of California, Berkeley, November 14.

Bird, S., Phansalkar, A., John, L.K., Mericas, A., Indukuru, R., 2007. Characterization of performance of SPEC CPU benchmarks on Intel's Core Microarchitecture based processor. In: Proceedings of 2007 SPEC Benchmark Workshop, January 21, 2007, Austin, TX.

Birman, M., Samuels, A., Chu, G., Chuk, T., Hu, L., McLeod, J., Barnes, J., 1990. Developing the WRL3170/3171 SPARC floating-point coprocessors. IEEE Micro 10 (1), 55–64.

Blackburn, M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 22–26, 2006, pp. 169–190.

Blaum, M., Brady, J., Bruck, J., Menon, J., 1994. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. In: Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago, IL, pp. 245–254.

Blaum, M., Brady, J., Bruck, J., Menon, J., 1995. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. IEEE Trans. Comput. 44 (2), 192–202.

Blaum, M., Bruck, J., Vardy, A., 1996. MDS array codes with independent parity symbols. IEEE Trans. Inf. Theory 42, 529–542.

Blaum, M., Brady, J., Bruck, J., Menon, J., Vardy, A., 2001. The EVENODD code and its generalization. In: Jin, H., Cortes, T., Buyya, R. (Eds.), High Performance Mass Storage and Parallel I/O: Technologies and Applications. Wiley-IEEE, New York, pp. 187–208.

Bloch, E., 1959. The engineering design of the Stretch computer. In: 1959 Proceedings of the Eastern Joint Computer Conference, December 1–3, 1959, Boston, MA, pp. 48–59.

Boddie, J.R., 2000. History of DSPs, www.lucent.com/micro/dsp/dsphist.html.

Boggs, D., Baktha, A., Hawkins, J., Marr, D.T., Miller, J.A., Roussel, P., et al., 2004. The Microarchitecture of the Intel Pentium 4 processor on 90 nm technology. Intel Technol. J. 8 (1), 7–23.

Bolt, K.M., 2005. Amazon sees sales rise, profit fall. Seattle Post-Intelligencer. http://seattlepi.nwsource.com/business/245943_techearns26.html.

Bordawekar, R., Bondhugula, U., Rao, R., 2010. Believe it or not!: multi-core CPUs can match GPU performance for a FLOP-intensive application! In: 19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010). Vienna, Austria, September 11–15, 2010, pp. 537–538.

Borg, A., Kessler, R.E., Wall, D.W., 1990. Generation and analysis of very long address traces. In: 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 270–279.

Bouknight, W.J., Deneberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., Slotnick, D.L., 1972. The Illiac IV system. Proc. IEEE 60 (4), 369–379. Also appears in Siewiorek, D.P., Bell, C.G., Newell, A. 1982. Computer Structures: Principles and Examples. McGraw-Hill, New York, pp. 306–316.

Brady, J.T., 1986. A theory of productivity in the creative process. IEEE Comput. Graph. Appl. 6 (5), 25–34.

Brain, M., 2000. Inside a Digital Cell Phone. www.howstuffworks.com/-inside-cellphone.htm.

Brandt, M., Brooks, J., Cahir, M., Hewitt, T., Lopez-Pineda, E., Sandness, D., 2000. The Benchmarker's Guide for Cray SV1 Systems. Cray Inc., Seattle, WA.

Brent, R.P., Kung, H.T., 1982. A regular layout for parallel adders. IEEE Trans. Comput. C-31, 260–264.

Brewer, E.A., Kuszmaul, B.C., 1994. How to get good performance from the CM-5 data network. In: Proceedings of Eighth International Parallel Processing Symposium, April 26–27, 1994, Cancun, Mexico.

Brin, S., Page, L., 1998. The anatomy of a large-scale hypertextual Web search engine. In: Proceedings of 7th International World Wide Web Conference, April 14–18, 1998, Brisbane, Queensland, Australia, pp. 107–117.

Brown, A., Patterson, D.A., 2000. Towards maintainability, availability, and growth benchmarks: a case study of software RAID systems. In: Proceedings of 2000 USENIX Annual Technical Conference, June 18–23, 2000, San Diego, CA.

Brunhaver, J.S., 2015. Design and optimization of a stencil engine (Ph.D. dissertation). Stanford University.

Bucher, I.Y., 1983. The computational speed of supercomputers. In: Proceedings of International Conference on Measuring and Modeling of Computer Systems (SIGMETRICS 1983), August 29–31, 1983, Minneapolis, MN, pp. 151–165.

Bucher, I.V., Hayes, A.H., 1980. I/O performance measurement on Cray-1 and CDC 7000 computers. In: Proceedings of Computer Performance Evaluation Users Group, 16th Meeting, NBS 500-65, pp. 245–254.

Bucholtz, W., 1962. Planning a Computer System: Project Stretch. McGraw-Hill, New York.

Burgess, N., Williams, T., 1995. Choices of operand truncation in the SRT division algorithm. IEEE Trans. Comput. 44 (7), 933–938.

Burkhardt III, H., Frank, S., Knobe, B., Rothnie, J., 1992. Overview of the KSR1 Computer System, Tech. Rep. KSR-TR-9202001. Kendall Square Research, Boston, MA.

Burks, A.W., Goldstine, H.H., von Neumann, J., 1946. Preliminary discussion of the logical design of an electronic computing instrument. Report to the U.S. Army Ordnance Department, p. 1; also appears in Papers of John von Neumann, Aspray, W., Burks, A. (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, CA, 1987, pp. 97–146.

Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B., 1997. Evidence-based static branch prediction using machine learning. ACM Trans. Program. Lang. Syst. 19 (1), 188–222.

Calder, B., Reinman, G., Tullsen, D.M., 1999. Selective value prediction. In: Proceedings of 26th Annual International Symposium on Computer Architecture (ISCA), May 2–4, 1999, Atlanta, GA.

Callahan, D., Dongarra, J., Levine, D., 1988. Vectorizing compilers: a test suite and results. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 12–17, 1988, Orland, FL, pp. 98–105.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Brown, S.D., Anderson, J.H., 2013. LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Trans. Embed. Comput. Syst. 13(2).

Canny, J., et al., 2015. Machine learning at the limit. In: IEEE International Conference on Big Data.

Cantin, J.F., Hill, M.D., 2001. Cache performance for selected SPEC CPU2000 benchmarks. www.jfred.org/cache-data.html.

Cantin, J.F., Hill, M.D., 2003. Cache performance for SPEC CPU2000 benchmarks, version 3.0. www.cs.wisc.edu/multifacet/misc/spec2000cache-data/index.html.

Carles, S., 2005. Amazon reports record Xmas season, top game picks. Gamasutra, December 27. http://www.gamasutra.com/php-bin/news_index.php?story=7630.

Carter, J., Rajamani, K., 2010. Designing energy-efficient servers and data centers. IEEE Comput. 43 (7), 76–78.

Case, R.P., Padegs, A., 1978. The architecture of the IBM System/370. Commun. ACM 21 (1), 73–96. Also appears in Siewiorek, D.P., Bell, C.G., Newell, A., 1982. Computer Structures: Principles and Examples. McGraw-Hill, New York, pp. 830–855.

Caulfield, A.M., Chung, E.S., Putnam, A., Haselman, H.A.J.F.M., Humphrey, S.H.M., Daniel, P.K.J.Y.K., Ovtcharov, L.T.M.K., Lanka, M.P.L.W.S., Burger, D.C.D., 2016. A cloud-scale acceleration architecture. In: MICRO Conference.

Censier, L., Feautrier, P., 1978. A new solution to coherence problems in multicache systems. IEEE Trans. Comput. C-27 (12), 1112–1118.

Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M., 1994. Scheduling and page migration for multiprocessor compute servers. In: Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 4–7, 1994, San Jose, CA, pp. 12–24.

Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.W., 1991. IMPACT: an architectural framework for multiple-instruction-issue processors. In: 18th Annual International Symposium on Computer Architecture (ISCA), May 27–30, 1991, Toronto, Canada, pp. 266–275.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E., 2006. Bigtable: a distributed storage system for structured data. In: Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06), November 6–8, 2006, Seattle, WA.

Chang, J., Meza, J., Ranganathan, P., Bash, C., Shah, A., 2010. Green server design: beyond operational energy to sustainability. In: Proceedings of Workshop on Power Aware Computing and Systems (HotPower'10), October 3, 2010, Vancouver, British Columbia.

Charlesworth, A.E., 1981. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. Computer 9, 18–27.

Charlesworth, A., 1998. Starfire: extending the SMP envelope. IEEE Micro 18 (1), 39–49.

Chen, T.C., 1980. Overlap and parallel processing. In: Stone, H. (Ed.), Introduction to Computer Architecture. Science Research Associates, Chicago, pp. 427–486.

Chen, S., 1983. Large-scale and high-speed multiprocessor system for scientific applications. In: Proceedings of NATO Advanced Research Workshop on High-Speed Computing, June 20–22, 1983, Jülich, West Germany. Also appears in Hwang, K. (Ed.), 1984. Superprocessors: design and applications, *IEEE* (August), 602–609.

Chen, P.M., Lee, E.K., 1995. Striping in a RAID level 5 disk array. In: Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 15–19, 1995, Ottawa, Canada, pp. 136–145.

Chen, P.M., Gibson, G.A., Katz, R.H., Patterson, D.A., 1990. An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890. In: Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 22–25, 1990, Boulder, CO.

Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A., 1994. RAID: high-performance, reliable secondary storage. ACM Comput. Surv. 26 (2), 145–188.

Chow, F.C., 1983. A Portable Machine-Independent Global Optimizer—Design and Measurements (Ph.D. thesis). Stanford University, Palo Alto, CA.

Chrysos, G.Z., Emer, J.S., 1998. Memory dependence prediction using store sets. In: Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA), July 3–14, 1998, Barcelona, Spain, pp. 142–153.

Clark, W.A., 1957. The Lincoln TX-2 computer development. In: Proceedings of Western Joint Computer Conference, February 26–28, 1957, Los Angeles, pp. 143–145.

Clark, D.W., 1983. Cache performance of the VAX-11/780. ACM Trans. Comput. Syst. 1 (1), 24–37.

Clark, D.W., 1987. Pipelining and performance in the VAX 8800 processor. In: Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 5–8, 1987, Palo Alto, CA, pp. 173–177.

Clark, J., 2014. Five Numbers That Illustrate the Mind-Bending Size of Amazon's Cloud. Bloomberg. https://www.bloomberg.com/news/2014-11-14/5-numbersthat-illustrate-the-mind-bending-size-of-amazon-s-cloud.html.

Clark, J., October 26, 2015. Google Turning Its Lucrative Web Search Over to AI Machines. Bloomberg Technology, www.bloomberg.com.

Clark, D.W., Emer, J.S., 1985. Performance of the VAX-11/780 translation buffer: simulation and measurement. ACM Trans. Comput. Syst. 3 (1), 31–62.

Clark, D., Levy, H., 1982. Measurement and analysis of instruction set use in the VAX-11/780. In: Proceedings of Ninth Annual International Symposium on Computer Architecture (ISCA), April 26–29, 1982, Austin, TX, pp. 9–17.

Clark, D., Strecker, W.D., 1980. Comments on 'the case for the reduced instruction set computer'. Comput. Architect. News 8 (6), 34–38.

Clark, B., Deshane, T., Dow, E., Evanchik, S., Finlayson, M., Herne, J., Neefe Matthews, J., 2004. Xen and the art of repeated research. In: Proceedings of USENIX Annual Technical Conference, June 27–July 2, 2004, pp. 135–144.

Clidaras, J., Johnson, C., Felderman, B., 2010. Private communication.

Climate Savers Computing Initiative, 2007. Efficiency Specs. http://www.climatesaverscomputing.org/.

Clos, C., 1953. A study of non-blocking switching networks. Bell Syst. Tech. J. 32 (2), 406–424.

Cloud, Bloomberg, n.d. https://www.bloomberg.com/news/2014-11-14/5-numbersthat-illustrate-the-mind-bending-size-of-amazon-s-cloud.html.

Cody, W.J., Coonen, J.T., Gay, D.M., Hanson, K., Hough, D., Kahan, W., Karpinski, R., Palmer, J., Ris, F.N., Stevenson, D., 1984. A proposed radix- and word-length independent standard for floating-point arithmetic. IEEE Micro 4 (4), 86–100.

Colwell, R.P., Steck, R., 1995. A 0.6 μm BiCMOS processor with dynamic execution. In: Proceedings of IEEE International Symposium on Solid State Circuits (ISSCC), February 15–17, 1995, San Francisco, pp. 176–177.

Colwell, R.P., Nix, R.P., O'Donnel, J.J., Papworth, D.B., Rodman, P.K., 1987. A VLIW architecture for a trace scheduling compiler. In: Proceedings of Second International

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 5–8, 1987, Palo Alto, CA, pp. 180–192.

Comer, D., 1993. Internetworking with TCP/IP, second ed. Prentice Hall, Englewood Cliffs, NJ.

Compaq Computer Corporation, 1999. Compiler Writer's Guide for the Alpha 21264, Order Number EC-RJ66A-TE, June, www1.support.compaq.com/alpha-tools/-documentation/current/21264_EV67/ec-rj66a-te_comp_writ_gde_for_alpha21264.pdf.

Conti, C., Gibson, D.H., Pitkowsky, S.H., 1968. Structural aspects of the System/360 Model 85. Part I. General organization. IBM Syst. J. 7 (1), 2–14.

Coonen, J., 1984. Contributions to a Proposed Standard for Binary Floating-Point Arithmetic (Ph.D. thesis). University of California, Berkeley.

Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., Sankar, S., 2004. Row-diagonal parity for double disk failure correction. In: Proceedings of 3rd USENIX Conference on File and Storage Technology (FAST'04), March 31–April 2, 2004, San Francisco.

Crawford, J., Gelsinger, P., 1988. Programming the 80386. Sybex Books, Alameda, CA.

Culler, D.E., Singh, J.P., Gupta, A., 1999. Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, San Francisco.

Curnow, H.J., Wichmann, B.A., 1976. A synthetic benchmark. Comput. J. 19 (1), 43–49.

Cvetanovic, Z., Kessler, R.E., 2000. Performance analysis of the Alpha 21264-based Compaq ES40 system. In: Proceedings of 27th Annual International Symposium on Computer Architecture (ISCA), June 10–14, 2000, Vancouver, Canada, pp. 192–202.

Dally, W.J., 1990. Performance analysis of *k*-ary *n*-cube interconnection networks. IEEE Trans. Comput. 39 (6), 775–785.

Dally, W.J., 1992. Virtual channel flow control. IEEE Trans. Parallel Distrib. Syst. 3 (2), 194–205.

Dally, W.J., 1999. Interconnect limited VLSI architecture. In: Proceedings of the International Interconnect Technology Conference, May 24–26, 1999, San Francisco.

Dally, W.J., 2002. Computer architecture is all about interconnect. In: Proceedings of the 8th International Symposium High Performance Computer Architecture.

Dally, W.J., 2016. High Performance Hardware for Machine Learning. Cadence Embedded Neural Network Summit, February 9, 2016. http://ip.cadence.com/uploads/presentations/1000AM_Dally_Cadence_ENN.pdf.

Dally, W.J., Seitz, C.I., 1986. The torus routing chip. Distrib. Comput. 1 (4), 187–196.

Dally, W.J., Towles, B., 2001. Route packets, not wires: on-chip interconnection networks. In: Proceedings of 38th Design Automation Conference, June 18–22, 2001, Las Vegas.

Dally, W.J., Towles, B., 2003. Principles and Practices of Interconnection Networks. Morgan Kaufmann, San Francisco.

Darcy, J.D., Gay, D., 1996. FLECKmarks: measuring floating point performance using a full IEEE compliant arithmetic benchmark. CS 252 class project, University of California, Berkeley. See http.CS.Berkeley.EDU/~darcy/Projects/cs252/.

Darley, H.M., et al., 1989. Floating Point/Integer Processor with Divide and Square Root Functions, U.S. Patent 4,878,190, October 31.

Davidson, E.S., 1971. The design and control of pipelined function generators. In: Proceedings of IEEE Conference on Systems, Networks, and Computers, January 19–21, 1971, Oaxtepec, Mexico, pp. 19–21.

Davidson, E.S., Thomas, A.T., Shar, L.E., Patel, J.H., 1975. Effective control for pipelined processors. In: Proceedings of IEEE COMPCON, February 25–27, 1975, San Francisco, pp. 181–184.

Davie, B.S., Peterson, L.L., Clark, D., 1999. Computer Networks: A Systems Approach, second ed. Morgan Kaufmann, San Francisco.

Dean, J., 2009. Designs, lessons and advice from building large distributed systems [keynote address]. In: Proceedings of 3rd ACM SIGOPS International Workshop on Large-Scale Distributed Systems and Middleware, Co-located with the 22nd ACM Symposium on Operating Systems Principles, October 11–14, 2009, Big Sky, Mont.

Dean, J., Barroso, L.A., 2013. The tail at scale. Commun. ACM 56 (2), 74–80.

Dean, J., Ghemawat, S., 2004. MapReduce: simplified data processing on large clusters. In: Proceedings of Operating Systems Design and Implementation (OSDI), December 6–8, 2004, San Francisco, CA, pp. 137–150.

Dean, J., Ghemawat, S., 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W., 2007. Dynamo: Amazon’s highly available key-value store. In: Proceedings of 21st ACM Symposium on Operating Systems Principles, October 14–17, 2007, Stevenson, WA.

Dehnert, J.C., Hsu, P.Y.-T., Bratt, J.P., 1989. Overlapped loop support on the Cydra 5. In: Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 3–6, 1989, Boston, MA, pp. 26–39.

Demmel, J.W., Li, X., 1994. Faster numerical algorithms via exception handling. IEEE Trans. Comput. 43 (8), 983–992.

Denehy, T.E., Bent, J., Popovici, F.I., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., 2004. Deconstructing storage arrays. In: Proceedings of 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 7–13, 2004, Boston, MA, pp. 59–71.

Desurvire, E., 1992. Lightwave communications: the fifth generation. Sci. Am. (Int. Ed.) 266 (1), 96–103.

Diep, T.A., Nelson, C., Shen, J.P., 1995. Performance evaluation of the PowerPC 620 microarchitecture. In: Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA), June 22–24, 1995, Santa Margherita, Italy.

Digital Semiconductor, 1996. Alpha Architecture Handbook, Version 3. Digital Press, Maynard, MA.

Ditzel, D.R., McLellan, H.R., 1987. Branch folding in the CRISP microprocessor: reducing the branch delay to zero. In: Proceedings of 14th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1987, Pittsburgh, PA, pp. 2–7.

Ditzel, D.R., Patterson, D.A., 1980. Retrospective on high-level language computer architecture. In: Proceedings of Seventh Annual International Symposium on Computer Architecture (ISCA), May 6–8, 1980, La Baule, France, pp. 97–104.

Doherty, W.J., Kelisky, R.P., 1979. Managing VM/CMS systems for user effectiveness. IBM Syst. J. 18 (1), 143–166.

Doherty, W.J., Thadhani, A.J., 1982. The economic value of rapid response time. IBM Report.

Dongarra, J.J., 1986. A survey of high performance processors. In: Proceedings of IEEE COMPCON, March 3–6, 1986, San Francisco, pp. 8–11.

Dongarra, J.J., Luszczek, P., Petitet, A., 2003. The LINPACK benchmark: past, present and future. Concurr. Comput. Pract. Exp. 15 (9), 803–820.

Dongarra, J., Sterling, T., Simon, H., Strohmaier, E., 2005. High-performance computing: clusters, constellations, MPPs, and future directions. Comput. Sci. Eng. 7 (2), 51–59.

Douceur, J.R., Bolosky, W.J., 1999. A large scale study of file-system contents. In: Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1–9, 1999, Atlanta, GA, pp. 59–69.

Douglas, J., 2005. Intel 8xx series and Paxville Xeon-MP microprocessors. In: Paper Presented at Hot Chips 17, August 14–16, 2005, Stanford University, Palo Alto, CA.

Duato, J., 1993. A new theory of deadlock-free adaptive routing in wormhole networks. IEEE Trans. Parallel Distrib. Syst. 4 (12), 1320–1331.

Duato, J., Pinkston, T.M., 2001. A general theory for deadlock-free adaptive routing using a mixed set of resources. IEEE Trans. Parallel Distrib. Syst. 12 (12), 1219–1235.

Duato, J., Yalamanchili, S., Ni, L., 2003. Interconnection Networks: An Engineering Approach, 2nd printing Morgan Kaufmann, San Francisco.

Duato, J., Johnson, I., Flich, J., Naven, F., Garcia, P., Nachiondo, T., 2005a. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In: Proceedings of 11th International Symposium on High-Performance Computer Architecture, February 12–16, 2005, San Francisco.

Duato, J., Lysne, O., Pang, R., Pinkston, T.M., 2005b. Part I: a theory for deadlock-free dynamic reconfiguration of interconnection networks. IEEE Trans. Parallel Distrib. Syst. 16 (5), 412–427.

Dubois, M., Scheurich, C., Briggs, F., 1988. Synchronization, coherence, and event ordering. IEEE Comput. 21 (2), 9–21.

Dunigan, W., Vetter, K., White, K., Worley, P., 2005. Performance evaluation of the Cray X1 distributed shared memory architecture. IEEE Micro, 30–40.

Eden, A., Mudge, T., 1998. The YAGS branch prediction scheme. In: Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture, November 30–December 2, 1998, Dallas, TX, pp. 69–80.

Edmondson, J.H., Rubinfield, P.I., Preston, R., Rajagopalan, V., 1995. Superscalar instruction execution in the 21164 Alpha microprocessor. IEEE Micro 15 (2), 33–43.

Eggers, S., 1989. Simulation Analysis of Data Sharing in Shared Memory Multiprocessors (Ph.D. thesis). University of California, Berkeley.

Elder, J., Gottlieb, A., Kruskal, C.K., McAuliffe, K.P., Randolph, L., Snir, M., Teller, P., Wilson, J., 1985. Issues related to MIMD shared-memory computers: the NYU ultracomputer approach. In: Proceedings of 12th Annual International Symposium on Computer Architecture (ISCA), June 17–19, 1985, Boston, MA, pp. 126–135.

Ellis, J.R., 1986. Bulldog: A Compiler for VLIW Architectures. MIT Press, Cambridge, MA.

Emer, J.S., Clark, D.W., 1984. A characterization of processor performance in the VAX-11/780. In: Proceedings of 11th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1984, Ann Arbor, MI, pp. 301–310.

Enriquez, P., 2001. What happened to my dial tone? A study of FCC service disruption reports. In: Poster, Richard Tapia Symposium on the Celebration of Diversity in Computing, October 18–20, Houston, TX.

Erlichson, A., Nuckolls, N., Chesson, G., Hennessy, J.L., 1996. SoftFLASH: analyzing the performance of clustered distributed virtual shared memory. In: Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1–5, 1996, Cambridge, MA, pp. 210–220.

Esmaeilzadeh, H., Cao, T., Xi, Y., Blackburn, S.M., McKinley, K.S., 2011. Looking back on the language and hardware revolution: measured power, performance, and scaling. In: Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 5–11, 2011, Newport Beach, CA.

Esmaeilzadeh, H., Blem, E., St Amant, R., Sankaralingam, K., Burger, D., 2012. Power limitations and dark silicon challenge the future of multicore. ACM Trans. Comput. Syst. 30 (3), 115–138.

Evers, M., Patel, S.J., Chappell, R.S., Patt, Y.N., 1998. An analysis of correlation and predictability: what makes two-level branch predictors work. In: Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA), July 3–14, 1998, Barcelona, Spain, pp. 52–61.

Fabry, R.S., 1974. Capability based addressing. Commun. ACM 17 (7), 403–412.

Falsafi, B., Wood, D.A., 1997. Reactive NUMA: a design for unifying S-COMA and CC-NUMA. In: Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO, pp. 229–240.

Fan, X., Weber, W., Barroso, L.A., 2007. Power provisioning for a warehouse-sized computer. In: Proceedings of 34th Annual International Symposium on Computer Architecture (ISCA), June 9–13, 2007, San Diego, CA.

Farkas, K.I., Jouppi, N.P., 1994. Complexity/performance trade-offs with non-blocking loads. In: Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago.

Farkas, K.I., Jouppi, N.P., Chow, P., 1995. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In: Proceedings of First IEEE Symposium on High-Performance Computer Architecture, January 22–25, 1995, Raleigh, NC, pp. 78–89.

Farkas, K.I., Chow, P., Jouppi, N.P., Vranesic, Z., 1997. Memory-system design considerations for dynamically-scheduled processors. In: Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO, pp. 133–143.

Fazio, D., 1987. It's really much more fun building a supercomputer than it is simply inventing one. In: Proceedings of IEEE COMPCON, February 23–27, 1987, San Francisco, pp. 102–105.

Fikes, A., 2010. Storage architecture and challenges. In: Google Faculty Summit.

Fisher, J.A., 1981. Trace scheduling: a technique for global microcode compaction. IEEE Trans. Comput. 30 (7), 478–490.

Fisher, J.A., 1983. Very long instruction word architectures and ELI-512. In: 10th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1982, Stockholm, Sweden, pp. 140–150.

Fisher, J.A., Freudenberger, S.M., 1992. Predicting conditional branches from previous runs of a program. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston, MA, pp. 85–95.

Fisher, J.A., Rau, B.R., 1993. J. Supercomput., January (special issue).

Fisher, J.A., Ellis, J.R., Ruttenberg, J.C., Nicolau, A., 1984. Parallel processing: a smart compiler and a dumb processor. In: Proceedings of SIGPLAN Conference on Compiler Construction, June 17–22, 1984, Montreal, Canada, pp. 11–16.

Flemming, P.J., Wallace, J.J., 1986. How not to lie with statistics: the correct way to summarize benchmarks results. Commun. ACM 29 (3), 218–221.

Flynn, M.J., 1966. Very high-speed computing systems. Proc. IEEE 54 (12), 1901–1909.

Forgie, J.W., 1957. The Lincoln TX-2 input-output system. In: Proceedings of Western Joint Computer Conference (February), Institute of Radio Engineers, Los Angeles, pp. 156–160.

Foster, C.C., Riseman, E.M., 1972. Percolation of code to enhance parallel dispatching and execution. IEEE Trans. Comput. C-21 (12), 1411–1415.

Frank, S.J., 1984. Tightly coupled multiprocessor systems speed memory access time. Electronics 57 (1), 164–169.

Freescale as part of i.MX31 Applications Processor, 2006. http://cache.freescale.com/files/32bit/doc/white_paper/IMX31MULTIWP.pdf.

Freiman, C.V., 1961. Statistical analysis of certain binary division algorithms. Proc. IRE 49 (1), 91–103.

Friesenborg, S.E., Wicks, R.J., 1985. DASD Expectations: The 3380, 3380-23, and MVS/XA, Tech. Bulletin GG22-9363-02. IBM Washington Systems Center, Gaithersburg, MD.

Fuller, S.H., Burr, W.E., 1977. Measurement and evaluation of alternative computer architectures. Computer 10 (10), 24–35.

Furber, S.B., 1996. ARM System Architecture. Addison-Wesley, Harlow, England. www.cs.man.ac.uk/amulet/publications/books/ARMsysArch.

Gagliardi, U.O., 1973. Report of workshop 4—software-related advances in computer hardware. In: Proceedings of Symposium on the High Cost of Software, September 17–19, 1973, Monterey, CA, pp. 99–120.

Gajski, D., Kuck, D., Lawrie, D., Sameh, A., 1983. CEDAR—a large scale multiprocessor. In: Proceedings of International Conference on Parallel Processing (ICPP), August, Columbus, Ohio, pp. 524–529.

Galal, S., Shacham, O., Brunhaver II, J.S., Pu, J., Vassiliev, A., Horowitz, M., 2013. FPU generator for design space exploration. In: 21st IEEE Symposium on Computer Arithmetic (ARITH).

Gallagher, D.M., Chen, W.Y., Mahlke, S.A., Gyllenhaal, J.C., Hwu, W.W., 1994. Dynamic memory disambiguation using the memory conflict buffer. In: Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 4–7, Santa Jose, CA, pp. 183–193.

Galles, M., 1996. Scalable pipelined interconnect for distributed endpoint routing: the SGI SPIDER chip. In: Proceedings of IEEE HOT Interconnects'96, August 15–17, 1996, Stanford University, Palo Alto, CA.

Game, M., Booker, A., 1999. CodePack code compression for PowerPC processors. MicroNews. 5 (1). www.chips.ibm.com/micronews/vol5_no1/codepack.html.

Gao, Q.S., 1993. The Chinese remainder theorem and the prime memory system. In: 20th Annual International Symposium on Computer Architecture (ISCA), May 16–19, 1993, San Diego, CA (Computer Architecture News 21:2 (May), pp. 337–340.

Gap, 2005. Gap Inc. Reports Third Quarter Earnings. http://gapinc.com/public/documents/PR_Q405EarningsFeb2306.pdf.

Gap, 2006. Gap Inc. Reports Fourth Quarter and Full Year Earnings. http://-gapinc.com/public/documents/Q32005PressRelease_Final22.pdf.

Garner, R., Agarwal, A., Briggs, F., Brown, E., Hough, D., Joy, B., Kleiman, S., Muchnick, S., Namjoo, M., Patterson, D., Pendleton, J., Tuck, R., 1988. Scalable processor architecture (SPARC). In: Proceedings of IEEE COMPCON, February 29–March 4, 1988, San Francisco, pp. 278–283.

Gebis, J., Patterson, D., 2007. Embracing and extending 20th-century instruction set architectures. IEEE Comput. 40 (4), 68–75.

Gee, J.D., Hill, M.D., Pnevmatikatos, D.N., Smith, A.J., 1993. Cache performance of the SPEC92 benchmark suite. IEEE Micro 13 (4), 17–27.

Gehringer, E.F., Siewiorek, D.P., Segall, Z., 1987. Parallel Processing: The Cm* Experience. Digital Press, Bedford, MA.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.L., 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of 17th Annual International Symposium on Computer Architecture (ISCA), May 28–31, 1990, Seattle, WA, pp. 15–26.

Gharachorloo, K., Gupta, A., Hennessy, J.L., 1992. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia.

Ghemawat, S., Gobioff, H., Leung, S.-T., 2003. The Google file system. In: Proceedings of 19th ACM Symposium on Operating Systems Principles, October 19–22, 2003, Bolton Landing, NY.

Gibson, D.H., 1967. Considerations in block-oriented systems design. AFIPS Conf. Proc. 30, 75–80.

Gibson, J.C., 1970. The Gibson mix, Rep. TR. 00.2043. IBM Systems Development Division, Poughkeepsie, NY (research done in 1959).

Gibson, G.A., 1992. In: Redundant Disk Arrays: Reliable, Parallel Secondary Storage. ACM Distinguished Dissertation Series, MIT Press, Cambridge, MA.

Gibson, J., Kunz, R., Ofelt, D., Horowitz, M., Hennessy, J., Heinrich, M., 2000. FLASH vs. (simulated) FLASH: Closing the simulation loop. In: Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), November 12–15, Cambridge, MA, pp. 49–58.

Glass, C.J., Ni, L.M., 1992. The Turn Model for adaptive routing. In: 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia.

Goldberg, I.B., 1967. 27 bits are not enough for 8-digit accuracy. Commun. ACM 10 (2), 105–106.

Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. Comput. Surv. 23 (1), 5–48.

Goldstein, S., 1987. Storage Performance—An Eight Year Outlook, Tech. Rep. TR 03.308-1. Santa Teresa Laboratory, IBM Santa Teresa Laboratory, San Jose, CA.

Goldstine, H.H., 1972. The Computer: From Pascal to von Neumann. Princeton University Press, Princeton, NJ.

González, A., Day, M., 2016. Amazon, Microsoft invest billions as computing shifts to cloud. The Seattle Times. http://www.seattletimes.com/business/technology/amazon-microsoft-invest-billions-as-computing-shifts-to-cloud/.

González, J., González, A., 1998. Limits of instruction level parallelism with data speculation. In: Proceedings of Vector and Parallel Processing (VECPAR) Conference, June 21–23, 1998, Porto, Portugal, pp. 585–598.

Goodman, J.R., 1983. Using cache memory to reduce processor memory traffic. In: Proceedings of 10th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1982, Stockholm, Sweden, pp. 124–131.

Goralski, W., 1997. SONET: A Guide to Synchronous Optical Network. McGraw-Hill, New York.

Gosling, J.B., 1980. Design of Arithmetic Units for Digital Computers. Springer-Verlag, New York.

Gray, J., 1990. A census of Tandem system availability between 1985 and 1990. IEEE Trans. Reliab. 39 (4), 409–418.

Gray, J. (Ed.), 1993. The Benchmark Handbook for Database and Transaction Processing Systems, second ed. Morgan Kaufmann, San Francisco.

Gray, J., 2006. Sort benchmark home page. http://sortbenchmark.org/.

Gray, J., Reuter, A., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco.

Gray, J., Siewiorek, D.P., 1991. High-availability computer systems. Computer 24 (9), 39–48.

Gray, J., van Ingen, C., 2005. Empirical Measurements of Disk Failure Rates and Error Rates, MSR-TR-2005-166. Microsoft Research, Redmond, WA.

Greenberg, A., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D., Patel, P., Sengupta, S., 2009. VL2: a scalable and flexible data center network. In: Proceedings of ACM SIGCOMM, August 17–21, 2009, Barcelona, Spain.

Grice, C., Kanellos, M., 2000. Cell phone industry at crossroads: go high or low? CNET News.technews.netscape.com/news/0-1004-201-2518386-0.html?tag=st.ne.1002.tgif.sf.

Groe, J.B., Larson, L.E., 2000. CDMA Mobile Radio Design. Artech House, Boston.

Gunther, K.D., 1981. Prevention of deadlocks in packet-switched data transport systems. IEEE Trans. Commun. 29 (4), 512–524.

Hagersten, E., Koster, M., 1998. WildFire: a scalable path for SMPs. In: Proceedings of Fifth International Symposium on High-Performance Computer Architecture, January 9–12, 1999, Orlando, FL.

Hagersten, E., Landin, A., Haridi, S., 1992. DDM—a cache-only memory architecture. IEEE Comput. 25 (9), 44–54.

Hamacher, V.C., Vranesic, Z.G., Zaky, S.G., 1984. Computer Organization, second ed. McGraw-Hill, New York.

Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M., 2010. Understanding sources of inefficiency in general-purpose chips. ACM SIGARCH Comput. Architect. News 38 (3), 37–47.

Hamilton, J., 2009. Data center networks are in my way. In: Paper Presented at the Stanford Clean Slate CTO Summit, October 23, 2009. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CleanSlateCTO2009.pdf.

Hamilton, J., 2010. Cloud computing economies of scale. In: Paper Presented at the AWS Workshop on Genomics and Cloud Computing, June 8, 2010, Seattle, WA. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_GenomicsCloud20100608.pdf.

Hamilton, J., 2014. AWS Innovation at Scale, AWS Re-invent conference. https://www.youtube.com/watch?v=JIQETrFC_SQ.

Hamilton, J., 2015. The Return to the Cloud. http://perspectives.mvdirona.com/2015/05/the-return-to-the-cloud//.

Hamilton, J., 2017. How Many Data Centers Needed World-Wide. http://perspectives.mvdirona.com/2017/04/how-many-data-centers-needed-worldwide/.

Hammerstrom, D., 1990. A VLSI architecture for high-performance, low-cost, on-chip learning. In: IJCNN International Joint Conference on Neural Networks.

Handy, J., 1993. The Cache Memory Book. Academic Press, Boston.

Hauck, E.A., Dent, B.A., 1968. Burroughs' B6500/B7500 stack mechanism. In: Proceedings of AFIPS Spring Joint Computer Conference, April 30–May 2, 1968, Atlantic City, NJ, pp. 245–251.

He, K., Zhang, X., Ren, S., Sun, J., 2016. Identity mappings in deep residual networks. Also in arXiv preprint arXiv:1603.05027.

Heald, R., Aingaran, K., Amir, C., Ang, M., Boland, M., Das, A., Dixit, P., Gouldsberry, G., Hart, J., Horel, T., Hsu, W.-J., Kaku, J., Kim, C., Kim, S., Klass, F., Kwan, H., Lo, R., McIntyre, H., Mehta, A., Murata, D., Nguyen, S., Pai, Y.-P., Patel, S., Shin, K., Tam, K., Vishwanthaiah, S., Wu, J., Yee, G., You, H., 2000. Implementation of third-generation SPARC V9 64-b microprocessor. In: ISSCC Digest of Technical Papers, pp. 412–413.

Heinrich, J., 1993. MIPS R4000 User's Manual. Prentice Hall, Englewood Cliffs, NJ.

Henly, M., McNutt, B., 1989. DASD I/O Characteristics: A Comparison of MVS to VM, Tech. Rep. TR 02.1550 (May). IBM General Products Division, San Jose, CA.

Hennessy, J., 1984. VLSI processor architecture. IEEE Trans. Comput. C-33 (11), 1221–1246.

Hennessy, J., 1985. VLSI RISC processors. VLSI Syst. Des. 6 (10), 22–32.

Hennessy, J., Jouppi, N., Baskett, F., Gill, J., 1981. MIPS: a VLSI processor architecture. In: CMU Conference on VLSI Systems and Computations. Computer Science Press, Rockville, MD.

Hewlett-Packard, 1994. PA-RISC 2.0 Architecture Reference Manual, third ed. Hewlett-Packard, Palo Alto, CA.

Hewlett-Packard, 1998. HP's '5NINES:5MINUTES' Vision Extends Leadership and Redefines High Availability in Mission-Critical Environments. www.future.enterprisecomputing.hp.com/ia64/news/5nines_vision_pr.html.

Hill, M.D., 1987. Aspects of Cache Memory and Instruction Buffer Performance (Ph.D. thesis). Tech. Rep. UCB/CSD 87/381. Computer Science Division, University of California, Berkeley.

Hill, M.D., 1988. A case for direct mapped caches. Computer 21 (12), 25–40.

Hill, M.D., 1998. Multiprocessors should support simple memory consistency models. IEEE Comput. 31 (8), 28–34.

Hillis, W.D., 1985. The Connection Multiprocessor. MIT Press, Cambridge, MA.

Hillis, W.D., Steele, G.L., 1986. Data parallel algorithms. Commun. ACM 29 (12), 1170–1183.

Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., Roussel, P., 2001. The microarchitecture of the Pentium 4 processor. Intel Technol. J.

Hintz, R.G., Tate, D.P., 1972. Control data STAR-100 processor design. In: Proceedings of IEEE COMPCON, September 12–14, 1972, San Francisco, pp. 1–4.

Hirata, H., Kimura, K., Nagamine, S., Mochizuki, Y., Nishimura, A., Nakase, Y., Nishizawa, T., 1992. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 136–145.

Hitachi, 1997. SuperH RISC Engine SH7700 Series Programming Manual. Hitachi, Santa Clara, CA. www.halsp.hitachi.com/tech_prod/.

Ho, R., Mai, K.W., Horowitz, M.A., 2001. The future of wires. In: Proc. of the IEEE, 89. 4, pp. 490–504.

Hoagland, A.S., 1963. Digital Magnetic Recording. Wiley, New York.

Hockney, R.W., Jesshope, C.R., 1988. Parallel Computers 2: Architectures, Programming and Algorithms. Adam Hilger, Ltd., Bristol, England.

Holland, J.H., 1959. A universal computer capable of executing an arbitrary number of subprograms simultaneously. Proc. East Joint Comput. Conf. 16, 108–113.

Holt, R.C., 1972. Some deadlock properties of computer systems. ACM Comput. Surv. 4 (3), 179–196.

Hölzle, U., 2010. Brawny cores still beat wimpy cores, most of the time. IEEE Micro 30, 4 (July/August).

Hopkins, M., 2000. A critical look at IA-64: massive resources, massive ILP, but can it deliver? Microprocessor Rep. February.

Hord, R.M., 1982. The Illiac-IV, The First Supercomputer. Computer Science Press, Rockville, MD.

Horel, T., Lauterbach, G., 1999. UltraSPARC-III: designing third-generation 64-bit performance. IEEE Micro 19 (3), 73–85.

Hospodor, A.D., Hoagland, A.S., et al., 1993. The changing nature of disk controllers. Proc. IEEE 81 (4), 586–594.

Hristea, C., Lenoski, D., Keen, J., 1997. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 16–21, 1997, San Jose, CA.

Hsu, P., 1994. Designing the TFP microprocessor. IEEE Micro 18(2).

Huang, M., Wu, D., Yu, C.H., Fang, Z., Interlandi, M., Condie, T., Cong, J., 2016. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, pp. 456–469.

Huck, J., et al., 2000. Introducing the IA-64 Architecture. IEEE Micro 20 (5), 12–23.

Hughes, C.J., Kaul, P., Adve, S.V., Jain, R., Park, C., Srinivasan, J., 2001. Variability in the execution of multimedia applications and implications for architecture. In: Proceedings of 28th Annual International Symposium on Computer Architecture (ISCA), June 30–July 4, 2001, Goteborg, Sweden, pp. 254–265.

Hwang, K., 1979. Computer Arithmetic: Principles, Architecture, and Design. Wiley, New York.

Hwang, K., 1993. Advanced Computer Architecture and Parallel Programming. McGraw-Hill, New York.

Hwu, W.-M., Patt, Y., 1986. HPSm, a high performance restricted data flow architecture having minimum functionality. In: Proceedings of 13th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1986, Tokyo, pp. 297–307.

Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.O., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M., 1993. The superblock: an effective technique for VLIW and superscalar compilation. J. Supercomput. 7 (1), 229–248.

Iandola, F., 2016. Exploring the Design Space of Deep Convolutional Neural Networks at Large Scale (Ph.D. dissertation). UC Berkeley.

IBM, 1982. The Economic Value of Rapid Response Time, GE20-0752-0. IBM, White Plains, NY, pp. 11–82.

IBM, 1990. The IBM RISC System/6000 processor. IBM J. Res. Dev. 34(1).

IBM, 1994. The PowerPC Architecture. Morgan Kaufmann, San Francisco.

IBM, 2005. Blue Gene. IBM J. Res. Dev. 49 (2/3) (Special issue).

IEEE, 1985. IEEE standard for binary floating-point arithmetic. SIGPLAN Notices 22 (2), 9–25.

IEEE, 2005. Intel virtualization technology, computer. IEEE Comput. Soc. 38 (5), 48–56.

IEEE 754-2008 Working Group, 2006. DRAFT Standard for Floating-Point Arithmetic 754-2008, https://doi.org/10.1109/IEEESTD.2008.4610935.

Ienne, P., Cornu, T., Kuhn, G., 1996. Special-purpose digital hardware for neural networks: an architectural survey. J. VLSI Signal Process. Syst. Signal Image Video Technol. 13(1).

Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB, Document No. 64402302, Imprimis, Dallas, TX.

InfiniBand Trade Association, 2001. InfiniBand Architecture Specifications Release 1.0.a. www.infinibandta.org.

Inoue, K., Ishihara, T., Murakami, K., 1999. Way-predicting set-associative cache for high performance and low energy consumption. In: Proc. 1999 International Symposium on Low Power Electronics and Design, ACM, pp. 273–275.

Intel, 2001. Using MMX Instructions to Convert RGB to YUV Color Conversion. cedar.intel.com/cgi-bin/ids.dll/content/content.jsp?cntKey=Legacy::irtm_AP548_9996&cntType=IDS_EDITORIAL.

Internet Retailer, 2005. The Gap launches a new site—after two weeks of downtime. Internet Retailer. http://www.internetretailer.com/2005/09/28/the-gap-launches-a-new-site-after-two-weeks-of-downtime.

Jain, R., 1991. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, New York.

Jantsch, A., Tenhunen, H. (Eds.), 2003. Networks on Chips. Kluwer Academic Publishers, The Netherlands.

Jimenez, D.A., Lin, C., 2001. Dynamic branch prediction with perceptrons. In: Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01). IEEE, Washington, DC, pp. 197–206.

Jimenez, D.A., Lin, C., 2002. Neural methods for dynamic branch prediction. ACM Trans. Comput. Syst. 20 (4), 369–397.

Johnson, M., 1990. Superscalar Microprocessor Design. Prentice Hall, Englewood Cliffs, NJ.

Jordan, H.F., 1983. Performance measurements on HEP—a pipelined MIMD computer. In: Proceedings of 10th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1982, Stockholm, Sweden, pp. 207–212.

Jordan, K.E., 1987. Performance comparison of large-scale scientific processors: scalar mainframes, mainframes with vector facilities, and supercomputers. Computer 20 (3), 10–23.

Jouppi, N.P., 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: Proceedings of 17th Annual International Symposium on Computer Architecture (ISCA), May 28–31, 1990, Seattle, WA, pp. 364–373.

Jouppi, N.P., 1998. Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In: 25 Years of the International Symposia on Computer Architecture (Selected Papers). ACM, New York, pp. 71–73.

Jouppi, N., 2016. Google supercharges machine learning tasks with TPU custom chip. https://cloudplatform.googleblog.com.

Jouppi, N.P., Wall, D.W., 1989. Available instruction-level parallelism for super-scalar and superpipelined processors. In: Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 3–6, 1989, Boston, pp. 272–282.

Jouppi, N.P., Wilton, S.J.E., 1994. Trade-offs in two-level on-chip caching. In: Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago, pp. 34–45.

Jouppi, N., Young, C., Patil, N., Patterson, D., Agrawal, G., et al., 2017. Datacenter performance analysis of a matrix processing unit. In: 44th International Symposium on Computer Architecture.

Kaeli, D.R., Emma, P.G., 1991. Branch history table prediction of moving target branches due to subroutine returns. In: Proceedings of 18th Annual International Symposium on Computer Architecture (ISCA), May 27–30, 1991, Toronto, Canada, pp. 34–42.

Kahan, W., 1968. 7094-II system support for numerical analysis, SHARE Secretarial Distribution SSD-159. Department of Computer Science, University of Toronto.

Kahan, J., 1990. On the advantage of the 8087's stack, unpublished course notes. Computer Science Division, University of California, Berkeley.

Kahaner, D.K., 1988. Benchmarks for 'real' programs. SIAM News. November.

Kahn, R.E., 1972. Resource-sharing computer communication networks. Proc. IEEE 60 (11), 1397–1407.

Kane, G., 1986. MIPS R2000 RISC Architecture. Prentice Hall, Englewood Cliffs, NJ.

Kane, G., 1996. PA-RISC 2.0 Architecture. Prentice Hall, Upper Saddle River, NJ.

Kane, G., Heinrich, J., 1992. MIPS RISC Architecture. Prentice Hall, Englewood Cliffs, NJ.

Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.Y., Brooks, D., 2015. Profiling a warehouse-scale computer. In: ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).

Karpathy, A., et al., 2014. Large-scale video classification with convolutional neural networks. CVPR.

Katz, R.H., Patterson, D.A., Gibson, G.A., 1989. Disk system architectures for high performance computing. Proc. IEEE 77 (12), 1842–1858.

Keckler, S.W., Dally, W.J., 1992. Processor coupling: integrating compile time and runtime scheduling for parallelism. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 202–213.

Keller, R.M., 1975. Look-ahead processors. ACM Comput. Surv. 7 (4), 177–195.

Keltcher, C.N., McGrath, K.J., Ahmed, A., Conway, P., 2003. The AMD Opteron processor for multiprocessor servers. IEEE Micro 23 (2), 66–76.

Kembel, R., 2000. Fibre channel: a comprehensive introduction. Internet Week. April.

Kermani, P., Kleinrock, L., 1979. Virtual cut-through: a new computer communication switching technique. Comput. Netw. 3, 267–286.

Kessler, R., 1999. The Alpha 21264 microprocessor. IEEE Micro 19 (2), 24–36.

Kilburn, T., Edwards, D.B.G., Lanigan, M.J., Sumner, F.H., 1962. One-level storage system. IRE Trans. Electron. Comput. EC-11, 223–235. Also appears in Siewiorek, D.P., Bell, C.G., Newell, A. 1982. Computer Structures: Principles and Examples. McGraw-Hill, New York. pp. 135–148.

Killian, E., 1991. MIPS R4000 technical overview–64 bits/100 MHz or bust. In: Hot Chips III Symposium Record, August 26–27, 1991, Stanford University, Palo Alto, CA. pp. 1.6–1.19.

Kim, M.Y., 1986. Synchronized disk interleaving. IEEE Trans. Comput. 35 (11), 978–988.

Kim, K., 2005. Technology for sub-50nm DRAM and NAND flash manufacturing. In: Electron Devices Meeting Technical Digest (December), pp. 323–326.

Kissell, K.D., 1997. MIPS16: High-density for the embedded market. In: Proceedings of Real Time Systems'97, June 15, 1997, Las Vegas, Nev. www.sgi.com/MIPS/arch/MIPS16/MIPS16.whitepaper.pdf.

Kitagawa, K., Tagaya, S., Hagihara, Y., Kanoh, Y., 2003. A hardware overview of SX-6 and SX-7 supercomputer. NEC Res. Dev. J. 44 (1), 2–7.

Knuth, D., 1981. second ed. The Art of Computer Programming, vol. II. Addison-Wesley, Reading, MA.

Kogge, P.M., 1981. The Architecture of Pipelined Computers. McGraw-Hill, New York.

Kohn, L., Fu, S.-W., 1989. A 1,000,000 transistor microprocessor. In: Proceedings of IEEE International Symposium on Solid State Circuits (ISSCC), February 15–17, 1989, New York, pp. 54–55.

Kohn, L., Margulis, N., 1989. Introducing the Intel i860 64-Bit Microprocessor. IEEE Micro 9 (4), 15–30.

Kontothanassis, L., Hunt, G., Stets, R., Hardavellas, N., Cierniak, M., Parthasarathy, S., Meira, W., Dwarkadas, S., Scott, M., 1997. VM-based shared memory on low-latency, remote-memory-access networks. In: Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO.

Koren, I., 1989. Computer Arithmetic Algorithms. Prentice Hall, Englewood Cliffs, NJ.

Kozyrakis, C., 2000. Vector IRAM: a media-oriented vector processor with embedded DRAM. In: Paper Presented at Hot Chips 12, August 13–15, 2000, Palo Alto, CA, pp. 13–15.

Kozyrakis, C., Patterson, D., 2002. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In: Proceedings of 35th Annual International Symposium on Microarchitecture (MICRO-35), November 18–22, 2002, Istanbul, Turkey.

Krizhevsky, A., Sutskever, I., Hinton, G., 2012. Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst.

Kroft, D., 1981. Lockup-free instruction fetch/prefetch cache organization. In: Proceedings of Eighth Annual International Symposium on Computer Architecture (ISCA), May 12–14, 1981, Minneapolis, MN, pp. 81–87.

Kroft, D., 1998. Retrospective: lockup-free instruction fetch/prefetch cache organization. In: 25 Years of the International Symposia on Computer Architecture (Selected Papers), ACM, New York, pp. 20–21.

Kuck, D., Budnik, P.P., Chen, S.-C., Lawrie, D.H., Towle, R.A., Strebendt, R.E., Davis Jr., E.W., Han, J., Kraska, P.W., Muraoka, Y., 1974. Measurements of parallelism in ordinary FORTRAN programs. Computer 7 (1), 37–46.

Kuhn, D.R., 1997. Sources of failure in the public switched telephone network. IEEE Comput. 30 (4), 31–36.

Kumar, A., 1997. The HP PA-8000 RISC CPU. IEEE Micro 17 (2), 27–32.

Kung, H.T., Leiserson, C.E., 1980. Algorithms for VLSI processor arrays. Introduction to VLSI systems.

Kunimatsu, A., Ide, N., Sato, T., Endo, Y., Murakami, H., Kamei, T., Hirano, M., Ishihara, F., Tago, H., Oka, M., Ohba, A., Yutaka, T., Okada, T., Suzuoki, M., 2000. Vector unit architecture for emotion synthesis. IEEE Micro 20 (2), 40–47.

Kunkel, S.R., Smith, J.E., 1986. Optimal pipelining in supercomputers. In: Proceedings of 13th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1986, Tokyo, pp. 404–414.

Kurose, J.F., Ross, K.W., 2001. Computer Networking: A Top-Down Approach Featuring the Internet. Addison-Wesley, Boston.

Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., Hennessy, J.L., 1994. The Stanford FLASH multiprocessor. In: Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago.

Lam, M., 1988. Software pipelining: an effective scheduling technique for VLIW processors. In: SIGPLAN Conference on Programming Language Design and Implementation, June 22–24, 1988, Atlanta, GA, pp. 318–328.

Lam, M.S., Wilson, R.P., 1992. Limits of control flow on parallelism. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 46–57.

Lam, M.S., Rothberg, E.E., Wolf, M.E., 1991. The cache performance and optimizations of blocked algorithms. In: Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 8–11, 1991, Santa Clara, CA. (SIGPLAN Notices 26:4 (April), 63–74).

Lambright, D., 2000. Experiences in measuring the reliability of a cache-based storage system. In: Proceedings of First Workshop on Industrial Experiences with Systems Software (WIESS 2000), Co-Located with the 4th Symposium on Operating Systems Design and Implementation (OSDI), October 22, 2000, San Diego, CA.

Lamport, L., 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. C-28 (9), 241–248.

Landstrom, B., 2014. The Cost of Downtime. http://www.interxion.com/blogs/2014/07/the-cost-of-downtime/.

Lang, W., Patel, J.M., Shankar, S., 2010. Wimpy node clusters: what about non-wimpy workloads? In: Proceedings of Sixth International Workshop on Data Management on New Hardware (DaMoN), June 7, Indianapolis, IN.

Laprie, J.-C., 1985. Dependable computing and fault tolerance: concepts and terminology. In: Proceedings of 15th Annual International Symposium on Fault-Tolerant Computing, June 19–21, 1985, Ann Arbor, Mich, pp. 2–11.

Larabel, M., 2016. Google Looks To Open Up StreamExecutor To Make GPGPU Programming Easier. Phoronix, March 10. https://www.phoronix.com/.

Larson, E.R., 1973. Findings of fact, conclusions of law, and order for judgment, File No. 4-67, Civ. 138, Honeywell v. Sperry-Rand and Illinois Scientific Development, U.S. District Court for the State of Minnesota, Fourth Division (October 19).

Laudon, J., Lenoski, D., 1997. The SGI Origin: a ccNUMA highly scalable server. In: Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO, pp. 241–251.

Laudon, J., Gupta, A., Horowitz, M., 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. In: Proceedings of Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 4–7, San Jose, CA, pp. 308–318.

Lauterbach, G., Horel, T., 1999. UltraSPARC-III: designing third generation 64-bit performance. IEEE Micro 19, 3 (May/June).

Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C., 1984. Quantitative System Performance: Computer System Analysis Using Queueing Network Models. Prentice Hall, Englewood Cliffs, NJ (Although out of print, it is available online at www.cs.washington.edu/homes/lazowska/qsp/).

Lebeck, A.R., Wood, D.A., 1994. Cache profiling and the SPEC benchmarks: a case study. Computer 27 (10), 15–26.

Lee, R., 1989. Precision architecture. Computer 22 (1), 78–91.

Lee, W.V., et al., 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In: Proceedings of 37th Annual International Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

Lee, Y., Waterman, A., Cook, H., Zimmer, B., Keller, B., Puggelli, A., Kwak, J., Jevtic, R., Bailey, S., Blagojevic, M., Chiu, P.-F., Avizienis, R., Richards, B., Bachrach, J., Patterson, D., Alon, E., Nikolic, B., Asanovic, K., 2016. An agile approach to building RISC-V microprocessors. IEEE Micro 36 (2), 8–20.

Leighton, F.T., 1992. Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, San Francisco.

Leiner, A.L., 1954. System specifications for the DYSEAC. J. ACM 1 (2), 57–81.

Leiner, A.L., Alexander, S.N., 1954. System organization of the DYSEAC. IRE Trans. Electron. Comput. 3 (1), 1–10.

Leiserson, C.E., 1985. Fat trees: universal networks for hardware-efficient supercomputing. IEEE Trans. Comput. C-34 (10), 892–901.

Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., Hennessy, J.L., 1990. The Stanford DASH multiprocessor. In: Proceedings of 17th Annual International Symposium on Computer Architecture (ISCA), May 28–31, 1990, Seattle, WA, pp. 148–159.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J.L., Horowitz, M.A., Lam, M., 1992. The Stanford DASH multiprocessor. IEEE Comput. 25 (3), 63–79.

Levy, H., Eckhouse, R., 1989. Computer Programming and Architecture: The VAX. Digital Press, Boston.

Lewis-Kraus, G., 2016. The Great A.I. Awakening. New York Times Magazine..

Li, K., 1988. IVY: a shared virtual memory system for parallel computing. In: Proceedings of 1988 International Conference on Parallel Processing. Pennsylvania State University Press, University Park, PA.

Li, S., Chen, K., Brockman, J.B., Jouppi, N., 2011. Performance Impacts of Non-blocking Caches in Out-of-order Processors. HP Labs Tech Report HPL-2011-65 (full text available at http://Library.hp.com/techpubs/2011/Hpl-2011-65.html).

Lim, K., Ranganathan, P., Chang, J., Patel, C., Mudge, T., Reinhardt, S., 2008. Understanding and designing new system architectures for emerging warehouse-computing environments. In: Proceedings of 35th Annual International Symposium on Computer Architecture (ISCA), June 21–25, 2008, Beijing, China.

Lincoln, N.R., 1982. Technology and design trade offs in the creation of a modern supercomputer. IEEE Trans. Comput. C-31 (5), 363–376.

Lindholm, T., Yellin, F., 1999. The Java Virtual Machine Specification, 2nd ed. Addison-Wesley, Reading, MA (Also available online at java.sun.com/docs/books/vmspec/).

Lipasti, M.H., Shen, J.P., 1996. Exceeding the dataflow limit via value prediction. In: Proceedings of 29th International Symposium on Microarchitecture, December 2–4, 1996, Paris, France.

Lipasti, M.H., Wilkerson, C.B., Shen, J.P., 1996. Value locality and load value prediction. In: Proceedings of Seventh Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1–5, 1996, Cambridge, MA, pp. 138–147.

Liptay, J.S., 1968. Structural aspects of the System/360 Model 85, Part II: The cache. IBM Syst. J. 7 (1), 15–21.

Lo, J., Eggers, S., Emer, J., Levy, H., Stamm, R., Tullsen, D., 1997. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. ACM Trans. Comput. Syst. 15 (2), 322–354.

Lo, J., Barroso, L., Eggers, S., Gharachorloo, K., Levy, H., Parekh, S., 1998. An analysis of database workload performance on simultaneous multithreaded processors. In: Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA), July 3–14, 1998, Barcelona, Spain, pp. 39–50.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L.A., Kozyrakis, C., 2014. Towards energy proportionality for large-scale latency-critical workloads. In: ACM/IEEE 41st Annual International Symposium on Computer Architecture (ISCA).

Loh, G.H., Hill, M.D., 2011. Efficiently enabling conventional block sizes for very large die-stacked DRAM caches. In: Proc. 44th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, pp. 454–464.

Lovett, T., Thakkar, S., 1988. The symmetry multiprocessor system. In: Proceedings of 1988 International Conference of Parallel Processing, University Park, PA, pp. 303–310.

Lubeck, O., Moore, J., Mendez, R., 1985. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. Computer 18 (12), 10–24.

Luk, C.-K., Mowry, T.C., 1999. Automatic compiler-inserted prefetching for pointer-based applications. IEEE Trans. Comput. 48 (2), 134–141.

Lunde, A., 1977. Empirical evaluation of some features of instruction set processor architecture. Commun. ACM 20 (3), 143–152.

Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D., 2005. Introduction to the HPC challenge benchmark suite. Lawrence Berkeley National Laboratory, Paper LBNL-57493 (April 25), repositories. cdlib.org/lbnl/LBNL-57493.

Maberly, N.C., 1966. Mastering Speed Reading. New American Library, New York.

Magenheimer, D.J., Peters, L., Pettis, K.W., Zuras, D., 1988. Integer multiplication and division on the HP precision architecture. IEEE Trans. Comput. 37 (8), 980–990.

Mahlke, S.A., Chen, W.Y., Hwu, W.-M., Rau, B.R., Schlansker, M.S., 1992. Sentinel scheduling for VLIW and superscalar processors. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston, pp. 238–247.

Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I., Hwu, W.W., 1995. A comparison of full and partial predicated execution support for ILP processors. In: Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA), June 22–24, 1995, Santa Margherita, Italy, pp. 138–149.

Major, J.B., 1989. Are queuing models within the grasp of the unwashed? In: Proceedings of International Conference on Management and Performance Evaluation of Computer Systems, December 11–15, 1989, Reno, Nev, pp. 831–839.

Markstein, P.W., 1990. Computation of elementary functions on the IBM RISC System/6000 processor. IBM J. Res. Dev. 34 (1), 111–119.

Mathis, H.M., Mercias, A.E., McCalpin, J.D., Eickemeyer, R.J., Kunkel, S.R., 2005. Characterization of the multithreading (SMT) efficiency in Power5. IBM J. Res. Dev. 49 (4/5), 555–564.

McCalpin, J., 2005. STREAM: Sustainable Memory Bandwidth in High Performance Computers. www.cs.virginia.edu/stream/.

McCalpin, J., Bailey, D., Takahashi, D., 2005. Introduction to the HPC Challenge Benchmark Suite, Paper LBNL-57493. Lawrence Berkeley National Laboratory, University of California, Berkeley, repositories.cdlib.org/lbnl/LBNL-57493.

McCormick, J., Knies, A., 2002. A brief analysis of the SPEC CPU2000 benchmarks on the Intel Itanium 2 processor. In: Paper Presented at Hot Chips 14, August 18–20, 2002, Stanford University, Palo Alto, CA.

McFarling, S., 1989. Program optimization for instruction caches. In: Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 3–6, 1989, Boston, pp. 183–191.

McFarling, S., 1993. Combining Branch Predictors, WRL Technical Note TN-36, Digital Western Research Laboratory, Palo Alto, CA.

McFarling, S., Hennessy, J., 1986. Reducing the cost of branches. In: Proceedings of 13th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1986, Tokyo, pp. 396–403.

McGhan, H., O'Connor, M., 1998. PicoJava: a direct execution engine for Java bytecode. Computer 31 (10), 22–30.

McKeeman, W.M., 1967. Language directed computer design. In: Proceedings of AFIPS Fall Joint Computer Conference, November 14–16, 1967, Washington, DC, pp. 413–417.

McMahon, F.M., 1986. The Livermore FORTRAN Kernels: A Computer Test of Numerical Performance Range, Tech. Rep. UCRL-55745. Lawrence Livermore National Laboratory, University of California, Livermore.

McNairy, C., Soltis, D., 2003. Itanium 2 processor microarchitecture. IEEE Micro 23 (2), 44–55.

Mead, C., Conway, L., 1980. Introduction to VLSI Systems. Addison-Wesley, Reading, MA.

Mellor-Crummey, J.M., Scott, M.L., 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9 (1), 21–65.

Menabrea, L.F., 1842. Sketch of the analytical engine invented by Charles Babbage. Bibliothèque Universelle de Genève. 82.

Menon, A., Renato Santos, J., Turner, Y., Janakiraman, G., Zwaenepoel, W., 2005. Diagnosing performance overheads in the xen virtual machine environment. In: Proceedings of First ACM/USENIX International Conference on Virtual Execution Environments, June 11–12, 2005, Chicago, pp. 13–23.

Merlin, P.M., Schweitzer, P.J., 1980. Deadlock avoidance in store-and-forward networks. Part I. Store-and-forward deadlock. IEEE Trans. Commun. 28 (3), 345–354.

Metcalfe, R.M., 1993. Computer/network interface design: lessons from Arpanet and Ethernet. IEEE J. Sel. Area. Commun. 11 (2), 173–180.

Metcalfe, R.M., Boggs, D.R., 1976. Ethernet: distributed packet switching for local computer networks. Commun. ACM 19 (7), 395–404.

Metropolis, N., Howlett, J., Rota, G.C. (Eds.), 1980. A History of Computing in the Twentieth Century. Academic Press, New York.

Meyer, R.A., Seawright, L.H., 1970. A virtual machine time sharing system. IBM Syst. J. 9 (3), 199–218.

Meyers, G.J., 1978. The evaluation of expressions in a storage-to-storage architecture. Comput. Architect. News 7 (3), 20–23.

Meyers, G.J., 1982. Advances in Computer Architecture, second ed. Wiley, New York.

Micron, 2004. Calculating Memory System Power for DDR2. http://download.micron.com/pdf/pubs/designline/dl1Q04.pdf.

Micron, 2006. The Micron System-Power Calculator. http://www.micron.com/-systemcalc.

MIPS, 1997. MIPS16 Application Specific Extension Product Description. www.sgi.com/MIPS/arch/MIPS16/mips16.pdf.

Miranker, G.S., Rubenstein, J., Sanguinetti, J., 1988. Squeezing a Cray-class supercomputer into a single-user package. In: Proceedings of IEEE COMPCON, February 29–March 4, 1988, San Francisco, pp. 452–456.

Mitchell, D., 1989. The transputer: the time is now. Comput. Des. (RISC suppl.) 40–41.

Mitsubishi, 1996. Mitsubishi 32-Bit Single Chip Microcomputer M32R Family Software Manual. Mitsubishi, Cypress, CA.

Miura, K., Uchida, K., 1983. FACOM vector processing system: VP100/200. In: Proceedings of NATO Advanced Research Workshop on High-Speed Computing, June 20–22, 1983, Jülich, West Germany. Also appears in Hwang, K. (Ed.), 1984. Superprocessors: Design and Applications. IEEE (August), pp. 59–73.

Miya, E.N., 1985. Multiprocessor/distributed processing bibliography. Comput. Architect. News 13 (1), 27–29.

Money, M.S.N., 2005. Amazon Shares Tumble after Rally Fizzles. http://moneycentral.msn.com/content/CNBCTV/Articles/Dispatches/P133695.asp.

Montoye, R.K., Hokenek, E., Runyon, S.L., 1990. Design of the IBM RISC System/6000 floating-point execution. IBM J. Res. Dev. 34 (1), 59–70.

Moore, G.E., 1965. Cramming more components onto integrated circuits. Electronics 38 (8), 114–117.

Moore, B., Padegs, A., Smith, R., Bucholz, W., 1987. Concepts of the System/370 vector architecture. In: 14th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1987, Pittsburgh, PA, pp. 282–292.

Morgan, T., 2014. A rare peek into the massive scale of AWS. Enterprise Tech. https://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/.

Morgan, T., 2016. How long can AWS keep climbing its steep growth curve? https://www.nextplatform.com/2016/02/01/how-long-can-aws-keep-climbingits-steep-growth-curve/.

Morse, S., Ravenal, B., Mazor, S., Pohlman, W., 1980. Intel microprocessors—8080 to 8086. Computer 13, 10.

Moshovos, A., Sohi, G.S., 1997. Streamlining inter-operation memory communication via data dependence prediction. In: Proceedings of 30th Annual International Symposium on Microarchitecture, December 1–3, Research Triangle Park, NC, pp. 235–245.

Moshovos, A., Breach, S., Vijaykumar, T.N., Sohi, G.S., 1997. Dynamic speculation and synchronization of data dependences. In: 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO.

Moussouris, J., Crudele, L., Freitas, D., Hansen, C., Hudson, E., Przybylski, S., Riordan, T., Rowen, C., 1986. A CMOS RISC processor with integrated system functions. In: Proceedings of IEEE COMPCON, March 3–6, 1986, San Francisco, p. 191.

Mowry, T.C., Lam, S., Gupta, A., 1992. Design and evaluation of a compiler algorithm for prefetching. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston (SIGPLAN Notices 27:9 (September), pp. 62–73.

Muchnick, S.S., 1988. Optimizing compilers for SPARC. Sun Technol. 1 (3), 64–77.

Mueller, M., Alves, L.C., Fischer, W., Fair, M.L., Modi, I., 1999. RAS strategy for IBM S/390 G5 and G6. IBM J. Res. Dev. 43 (5-6), 875–888.

Mukherjee, S.S., Weaver, C., Emer, J.S., Reinhardt, S.K., Austin, T.M., 2003. Measuring architectural vulnerability factors. IEEE Micro 23 (6), 70–75.

Murphy, B., Gent, T., 1995. Measuring system and software reliability using an automated data collection process. Qual. Reliab. Eng. Int. 11 (5), 341–353.

Myer, T.H., Sutherland, I.E., 1968. On the design of display processors. Commun. ACM 11 (6), 410–414.

Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., Rowstron, A., 2009. Migrating server storage to SSDs: analysis of trade-offs. In: Proceedings of 4th ACM European Conference on Computer Systems, April 1–3, 2009, Nuremberg, Germany.

National Research Council, 1997. The Evolution of Untethered Communications. Computer Science and Telecommunications Board, National Academy Press, Washington, DC.

National Storage Industry Consortium, 1998. Tape Roadmap. www.nsic.org.

Nelson, V.P., 1990. Fault-tolerant computing: fundamental concepts. Computer 23 (7), 19–25.

Ngai, T.-F., Irwin, M.J., 1985. Regular, area-time efficient carry-lookahead adders. In: Proceedings of Seventh IEEE Symposium on Computer Arithmetic, June 4–6, 1985, University of Illinois, Urbana, pp. 9–15.

Nicolau, A., Fisher, J.A., 1984. Measuring the parallelism available for very long instruction word architectures. IEEE Trans. Comput. C33 (11), 968–976.

Nielsen, M., 2016. Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/.

Nikhil, R.S., Papadopoulos, G.M., Arvind, 1992. *T: a multithreaded massively parallel architecture. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 156–167.

Noordergraaf, L., van der Pas, R., 1999. Performance experiences on Sun's WildFire prototype. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 13–19, 1999, Portland, Ore.

Nvidia, 2016. Tesla GPU Accelerators For Servers. http:// www.nvidia.com/object/tesla-servers.html.

Nyberg, C.R., Barclay, T., Cvetanovic, Z., Gray, J., Lomet, D., 1994. AlphaSort: a RISC machine sort. In: Proceedings of ACM SIGMOD, May 24–27, 1994, Minneapolis, Minn.

Oka, M., Suzuoki, M., 1999. Designing and programming the emotion engine. IEEE Micro 19 (6), 20–28.

Okada, S., Okada, S., Matsuda, Y., Yamada, T., Kobayashi, A., 1999. System on a chip for digital still camera. IEEE Trans. Consum. Electron. 45 (3), 584–590.

Oliker, L., Canning, A., Carter, J., Shalf, J., Ethier, S., 2004. Scientific computations on modern parallel vector systems. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 6–12, 2004, Pittsburgh, Penn, p. 10.

Olofsson, A., 2011. Debunking the myth of the $100M ASIC. EE Times. http://www.eetimes.com/author.asp?section_id=36&doc_id=1266014.

Ovtcharov, K., Ruwase, O., Kim, J.Y., Fowers, J., Strauss, K., Chung, E.S., 2015a. Accelerating deep convolutional neural networks using specialized hardware. Microsoft Research Whitepaper. https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

Ovtcharov, K., Ruwase, O., Kim, J.Y., Fowers, J., Strauss, K., Chung, E.S., 2015b. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In: 2015 IEEE Hot Chips 27 Symposium.

Pabst, T., 2000. Performance Showdown at 133 MHz FSB—The Best Platform for Coppermine. www6.tomshardware.com/mainboard/00q1/000302/.

Padua, D., Wolfe, M., 1986. Advanced compiler optimizations for supercomputers. Commun. ACM 29 (12), 1184–1201.

Palacharla, S., Kessler, R.E., 1994. Evaluating stream buffers as a secondary cache replacement. In: Proceedings of 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago, pp. 24–33.

Palmer, J., Morse, S., 1984. The 8087 Primer. John Wiley & Sons, New York, p. 93.

Pan, S.-T., So, K., Rameh, J.T., 1992. Improving the accuracy of dynamic branch prediction using branch correlation. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston, pp. 76–84.

Partridge, C., 1994. Gigabit Networking. Addison-Wesley, Reading, MA.

Patterson, D., 1985. Reduced instruction set computers. Commun. ACM 28 (1), 8–21.

Patterson, D., 2004. Latency lags bandwidth. Commun. ACM 47 (10), 71–75.

Patterson, D.A., Ditzel, D.R., 1980. The case for the reduced instruction set computer. ACM SIGARCH Comput. Architect. News 8 (6), 25–33.

Patterson, D.A., Hennessy, J.L., 2004. Computer Organization and Design: The Hardware/Software Interface, third ed. Morgan Kaufmann, San Francisco.

Patterson, D., Nikolić, B., 7/25/2015, Agile Design for Hardware, Parts I, II, and III. EE Times, http://www.eetimes.com/author.asp?doc_id=1327239.

Patterson, D.A., Garrison, P., Hill, M., Lioupis, D., Nyberg, C., Sippel, T., Van Dyke, K., 1983. Architecture of a VLSI instruction cache for a RISC. In: 10th Annual International Conference on Computer Architecture Conf. Proc., June 13–16, 1983, Stockholm, Sweden, pp. 108–116.

Patterson, D.A., Gibson, G.A., Katz, R.H., 1987. A Case for Redundant Arrays of Inexpensive Disks (RAID), Tech. Rep. UCB/CSD 87/391, University of California, Berkeley. Also appeared in Proc. ACM SIGMOD, June 1–3, 1988, Chicago, pp. 109–116.

Pavan, P., Bez, R., Olivo, P., Zanoni, E., 1997. Flash memory cells—an overview. Proc. IEEE 85 (8), 1248–1271.

Peh, L.S., Dally, W.J., 2001. A delay model and speculative architecture for pipe-lined routers. In: Proceedings of 7th International Symposium on High-Performance Computer Architecture, January 22–24, 2001, Monterrey, Mexico.

Peng, V., Samudrala, S., Gavrielov, M., 1987. On the implementation of shifters, multipliers, and dividers in VLSI floating point units. In: Proceedings of 8th IEEE Symposium on Computer Arithmetic, May 19–21, 1987, Como, Italy, pp. 95–102.

Pfister, G.F., 1998. In Search of Clusters, second ed. Prentice Hall, Upper Saddle River, NJ.

Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfekder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., Weiss, J., 1985. The IBM research parallel processor prototype (RP3): introduction and architecture. In: Proceedings of 12th Annual International Symposium on Computer Architecture (ISCA), June 17–19, 1985, Boston, MA, pp. 764–771.

Pinheiro, E., Weber, W.D., Barroso, L.A., 2007. Failure trends in a large disk drive population. In: Proceedings of 5th USENIX Conference on File and Storage Technologies (FAST ’07), February 13–16, 2007, San Jose, CA.

Pinkston, T.M., 2004. Deadlock characterization and resolution in interconnection networks. In: Zhu, M.C., Fanti, M.P. (Eds.), Deadlock Resolution in Computer-Integrated Systems. CRC Press, Boca Raton, FL, pp. 445–492.

Pinkston, T.M., Shin, J., 2005. Trends toward on-chip networked microsystems. Int. J. High Perform. Comput. Netw. 3 (1), 3–18.

Pinkston, T.M., Warnakulasuriya, S., 1997. On deadlocks in interconnection networks. In: 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO.

Pinkston, T.M., Benner, A., Krause, M., Robinson, I., Sterling, T., 2003. InfiniBand: the ‘de facto’ future standard for system and local area networks or just a scalable replacement for PCI buses?”. Cluster Comput. 6 (2), 95–104 (Special issue on communication architecture for clusters).

Postiff, M.A., Greene, D.A., Tyson, G.S., Mudge, T.N., 1999. The limits of instruction level parallelism in SPEC95 applications. Comput. Architect. News 27 (1), 31–40.

Prabhakar, R., Koeplinger, D., Brown, K.J., Lee, H., De Sa, C., Kozyrakis, C., Olukotun, K., 2016. Generating configurable hardware from parallel patterns. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, pp. 651–665.

Prakash, T.K., Peng, L., 2008. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor. ISAST Trans. Comput. Softw. Eng. 2 (1), 36–41.

Przybylski, S.A., 1990. Cache Design: A Performance-Directed Approach. Morgan Kaufmann, San Francisco.

Przybylski, S.A., Horowitz, M., Hennessy, J.L., 1988. Performance trade-offs in cache design. In: 15th Annual International Symposium on Computer Architecture, May 30–June 2, 1988, Honolulu, Hawaii, pp. 290–298.

Puente, V., Beivide, R., Gregorio, J.A., Prellezo, J.M., Duato, J., Izu, C., 1999. Adaptive bubble router: a design to improve performance in torus networks. In: Proceedings of the 28th International Conference on Parallel Processing, September 21–24, 1999, Aizu-Wakamatsu, Fukushima, Japan.

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In: 41st International Symposium on Computer Architecture.

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2015. A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro. 35(3).

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2016. A reconfigurable fabric for accelerating large-scale datacenter services. Commun. ACM. 59 (11), 114–122.

Qadeer, W., Hameed, R., Shacham, O., Venkatesan, P., Kozyrakis, C., Horowitz, M.A., 2015. Convolution engine: balancing efficiency & flexibility in specialized computing. Commun. ACM 58(4).

Qureshi, M.K., Loh, G.H., 2012. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In: Proc. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, pp. 235–246.

Radin, G., 1982. The 801 minicomputer. In: Proceedings of Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 1–3, 1982, Palo Alto, CA, pp. 39–47.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S., 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Not. 48 (6), 519–530.

Ramacher, U., Beichter, J., Raab, W., Anlauf, J., Bruels, N., Hachmann, A., Wesseling, M., 1991. Design of a 1st generation neurocomputer. VLSI Design of Neural Networks. Springer, USA.

Ramamoorthy, C.V., Li, H.F., 1977. Pipeline architecture. ACM Comput. Surv. 9 (1), 61–102.

Ranganathan, P., Leech, P., Irwin, D., Chase, J., 2006. Ensemble-level power management for dense blade servers. In: Proceedings of 33rd Annual International Symposium on Computer Architecture (ISCA), June 17–21, 2006, Boston, MA, pp. 66–77.

Rau, B.R., 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of 27th Annual International Symposium on Microarchitecture, November 30–December 2, 1994, San Jose, CA, pp. 63–74.

Rau, B.R., Fisher, J.A., 1993. Instruction-level parallelism. J. Supercomput. 235, Springer Science & Business Media.

Rau, B.R., Glaeser, C.D., Picard, R.L., 1982. Efficient code generation for horizontal architectures: compiler techniques and architectural support. In: Proceedings of Ninth Annual International Symposium on Computer Architecture (ISCA), April 26–29, 1982, Austin, TX, pp. 131–139.

Rau, B.R., Yen, D.W.L., Yen, W., Towle, R.A., 1989. The Cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs. IEEE Comput. 22 (1), 12–34.

Reddi, V.J., Lee, B.C., Chilimbi, T., Vaid, K., 2010. Web search using mobile cores: quantifying and mitigating the price of efficiency. In: Proceedings of 37th Annual International Symposium on Computer Architecture (ISCA), June 19–23, 2010, Saint-Malo, France.

Redmond, K.C., Smith, T.M., 1980. Project Whirlwind—The History of a Pioneer Computer. Digital Press, Boston.

Reinhardt, S.K., Larus, J.R., Wood, D.A., 1994. Tempest and typhoon: user-level shared memory. In: 21st Annual International Symposium on Computer Architecture (ISCA), April 18–21, 1994, Chicago, pp. 325–336.

Reinman, G., Jouppi, N.P., 1999. Extensions to CACTI. research.compaq.com/wrl/people/jouppi/CACTI.html.

Rettberg, R.D., Crowther, W.R., Carvey, P.P., Towlinson, R.S., 1990. The Monarch parallel processor hardware design. IEEE Comput. 23 (4), 18–30.

Riemens, A., Vissers, K.A., Schutten, R.J., Sijstermans, F.W., Hekstra, G.J., La Hei, G.D., 1999. Trimedia CPU64 application domain and benchmark suite. In: Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'99), October 10–13, 1999, Austin, TX, pp. 580–585.

Riseman, E.M., Foster, C.C., 1972. Percolation of code to enhance parallel dispatching and execution. IEEE Trans. Comput. C-21 (12), 1411–1415.

Robin, J., Irvine, C., 2000. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In: Proceedings of USENIX Security Symposium, August 14–17, 2000, Denver, CO.

Robinson, B., Blount, L., 1986. The VM/HPO 3880-23 Performance Results, IBM Tech. Bulletin GG66-0247-00. IBM Washington Systems Center, Gaithersburg, MD.

Ropers, A., Lollman, H.W., Wellhausen, J., 1999. DSPstone: Texas Instruments TMS320C54x, Tech. Rep. IB 315 1999/9-ISS-Version 0.9. Aachen University of Technology, Aachen, Germany (www.ert.rwth-aachen.de/Projekte/Tools/coal/dspstone_c54x/index.html).

Rosenblum, M., Herrod, S.A., Witchel, E., Gupta, A., 1995. Complete computer simulation: the SimOS approach. IEEE Parallel Distrib. Technol. 4 (3), 34–43.

Rowen, C., Johnson, M., Ries, P., 1988. The MIPS R3010 floating-point coprocessor. IEEE Micro 8 (3), 53–62.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., 2015. Imagenet large scale visual recognition challenge. Int. J. Comput. Vis. 115(3).

Russell, R.M., 1978. The Cray-1 processor system. Commun. ACM 21 (1), 63–72.

Rymarczyk, J., 1982. Coding guidelines for pipelined processors. In: Proceeding of Symposium Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 1–3, 1982, Palo Alto, CA, pp. 12–19.

Saavedra-Barrera, R.H., 1992. CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking (Ph.D. dissertation). University of California, Berkeley.

Salem, K., Garcia-Molina, H., 1986. Disk striping. In: Proceedings of 2nd International IEEE Conference on Data Engineering, February 5–7, 1986, Washington, DC, pp. 249–259.

Saltzer, J.H., Reed, D.P., Clark, D.D., 1984. End-to-end arguments in system design. ACM Trans. Comput. Syst. 2 (4), 277–288.

Samples, A.D., Hilfinger, P.N., 1988. Code Reorganization for Instruction Caches, Tech. Rep. UCB/CSD 88/447, University of California, Berkeley.

Santoro, M.R., Bewick, G., Horowitz, M.A., 1989. Rounding algorithms for IEEE multipliers. In: Proceedings of Ninth IEEE Symposium on Computer Arithmetic, September 6–8, Santa Monica, CA, pp. 176–183.

Satran, J., Smith, D., Meth, K., Sapuntzakis, C., Wakeley, M., Von Stamwitz, P., Haagens, R., Zeidner, E., Dalle Ore, L., Klein, Y., 2001. "iSCSI," IPS Working Group of IETF, Internet draft. www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt.

Saulsbury, A., Wilkinson, T., Carter, J., Landin, A., 1995. An argument for simple COMA. In: Proceedings of First IEEE Symposium on High-Performance Computer Architectures, January 22–25, 1995, Raleigh, NC, pp. 276–285.

Schneck, P.B., 1987. Superprocessor Architecture. Kluwer Academic Publishers, Norwell, MA.

Schroeder, B., Gibson, G.A., 2007. Understanding failures in petascale computers. J. Phys. Conf. Ser. 78 (1), 188–198.

Schroeder, B., Pinheiro, E., Weber, W.-D., 2009. DRAM errors in the wild: a large-scale field study. In: Proceedings of Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), June 15–19, 2009, Seattle, WA.

Schurman, E., Brutlag, J., 2009. The user and business impact of server delays. In: Proceedings of Velocity: Web Performance and Operations Conference, June 22–24, 2009, San Jose, CA.

Schwartz, J.T., 1980. Ultracomputers. ACM Trans. Program. Lang. Syst. 4 (2), 484–521.

Scott, N.R., 1985. Computer Number Systems and Arithmetic. Prentice Hall, Englewood Cliffs, NJ.

Scott, S.L., 1996. Synchronization and communication in the T3E multiprocessor. In: Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1–5, 1996, Cambridge, MA.

Scott, S.L., Goodman, J., 1994. The impact of pipelined channels on *k*-ary *n*-cube networks. IEEE Trans. Parallel Distrib. Syst. 5 (1), 1–16.

Scott, S.L., Thorson, G.M., 1996. The Cray T3E network: adaptive routing in a high performance 3D torus. In: Proceedings of IEEE HOT Interconnects '96, August 15–17, 1996, Stanford University, Palo Alto, CA, pp. 14–156.

Scranton, R.A., Thompson, D.A., Hunter, D.W., 1983. The Access Time Myth. Tech. Rep. RC 10197 (45223). IBM, Yorktown Heights, NY.

Seagate, 2000. Seagate Cheetah 73 Family: ST173404LW/LWV/LC/LCV Product Manual, vol. 1. Seagate, Scotts Valley, CA. www.seagate.com/support/disc/manuals/scsi/29478b.pdf.

Seitz, C.L., 1985. The Cosmic Cube (concurrent computing). Commun. ACM 28 (1), 22–33.

Senior, J.M., 1993. Optical Fiber Communications: Principles and Practice, second ed. Prentice Hall, Hertfordshire, UK.

Sergio Guadarrama, 2015. BVLC googlenet. https://github.com/BVLC/caffe/tree/ master/models/bvlc_googlenet.

Seznec, A., Michaud, P., 2006. A case for (partially) TAgged GEometric history length branch prediction. J. Instruction Level Parallel. 8, 1–23.

Shao, Y.S., Brooks, D., 2015. Research infrastructures for hardware accelerators. Synth. Lect. Comput. Architect. 10 (4), 1–99.

Sharangpani, H., Arora, K., 2000. Itanium processor microarchitecture. IEEE Micro 20 (5), 24–43.

Shurkin, J., 1984. Engines of the Mind: A History of the Computer. W.W. Norton, New York.

Shustek, L.J., 1978. Analysis and Performance of Computer Instruction Sets (Ph.D. dissertation). Stanford University, Palo Alto, CA.

Silicon Graphics, 1996. MIPS V Instruction Set. http://www.sgi.com/MIPS/arch/ISA5/#MIPSV_indx.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. Nature 529(7587).

Singh, J.P., Hennessy, J.L., Gupta, A., 1993. Scaling parallel programs for multiprocessors: methodology and examples. In: Computer, 2. 7, pp. 22–33.

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Eiichi Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., Vahdat, A., 2015. Jupiter rising: a decade of CLOS topologies and centralized control in Google's datacenter network. ACM SIGCOMM Comput. Commun. Rev. 45 (4), 183–197.

Sinharoy, B., Koala, R.N., Tendler, J.M., Eickemeyer, R.J., Joyner, J.B., 2005. POWER5 system microarchitecture. IBM J. Res. Dev. 49 (4–5), 505–521.

Sites, R., 1979. Instruction Ordering for the CRAY-1 Computer, Tech. Rep. 78-CS-023. Dept. of Computer Science, University of California, San Diego.

Sites, R.L. (Ed.), 1992. Alpha Architecture Reference Manual. Digital Press, Burlington, MA.

Sites, R.L., Witek, R. (Eds.), 1995. Alpha Architecture Reference Manual, second ed. Digital Press, Newton, MA.

Skadron, K., Clark, D.W., 1997. Design issues and tradeoffs for write buffers. In: Proceedings of Third International Symposium on High-Performance Computer Architecture, February 1–5, 1997, San Antonio, TX, pp. 144–155.

Skadron, K., Ahuja, P.S., Martonosi, M., Clark, D.W., 1999. Branch prediction, instruction-window size, and cache size: performance tradeoffs and simulation techniques. IEEE Trans. Comput. 48(11).

Slater, R., 1987. Portraits in Silicon. MIT Press, Cambridge, MA.

Slotnick, D.L., Borck, W.C., McReynolds, R.C., 1962. The Solomon computer. In: Proceedings of AFIPS Fall Joint Computer Conference, December 4–6, 1962, Philadelphia, PA, pp. 97–107.

Smith, B.J., 1978. A pipelined, shared resource MIMD computer. In: Proceedings of International Conference on Parallel Processing (ICPP), August, Bellaire, MI, pp. 6–8.

Smith, B.J., 1981a. Architecture and applications of the HEP multiprocessor system. Real Time Signal Process. IV 298, 241–248.

Smith, J.E., 1981b. A study of branch prediction strategies. In: Proceedings of Eighth Annual International Symposium on Computer Architecture (ISCA), May 12–14, 1981, Minneapolis, MN, pp. 135–148.

Smith, A.J., 1982a. Cache memories. Comput. Surv., 14, 3, pp. 473–530.

Smith, J.E., 1982b. Decoupled access/execute computer architectures. In: Proceedings of the 11th International Symposium on Computer Architecture.

Smith, J.E., 1984. Decoupled access/execute computer architectures. ACM Trans. Comput. Syst. 2 (4), 289–308.

Smith, J.E., 1988. Characterizing computer performance with a single number. Commun. ACM 31 (10), 1202–1206.

Smith, J.E., 1989. Dynamic instruction scheduling and the Astronautics ZS-1. Computer 22 (7), 21–35.

Smith, J.E., Goodman, J.R., 1983. A study of instruction cache organizations and replacement policies. In: Proceedings of 10th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1982, Stockholm, Sweden, pp. 132–137.

Smith, A., Lee, J., 1984. Branch prediction strategies and branch-target buffer design. Computer 17 (1), 6–22.

Smith, J.E., Pleszkun, A.R., 1988. Implementing precise interrupts in pipelined processors. IEEE Trans. Comput. 37 (5), 562–573. (This paper is based on an earlier paper that appeared in Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA), June 17–19, 1985, Boston, MA.

Smith, J.E., Dermer, G.E., Vanderwarn, B.D., Klinger, S.D., Rozewski, C.M., Fowler, D.L., Scidmore, K.R., Laudon, J.P., 1987. The ZS-1 central processor. In: Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 5–8, 1987, Palo Alto, CA, pp. 199–204.

Smith, M.D., Johnson, M., Horowitz, M.A., 1989. Limits on multiple instruction issue. In: Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 3–6, 1989, Boston, pp. 290–302.

Smith, M.D., Horowitz, M., Lam, M.S., 1992. Efficient superscalar performance through boosting. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston, pp. 248–259.

Smotherman, M., 1989. A sequencing-based taxonomy of I/O systems and review of historical machines. Comput. Architect. News 17 (5), 5–15. Reprinted in Computer Architecture Readings, Hill, M.D., Jouppi, N.P., Sohi, G.S. (Eds.), 1999. Morgan Kaufmann, San Francisco, pp. 451–461.

Sodani, A., Sohi, G., 1997. Dynamic instruction reuse. In: Proceedings of 24th Annual International Symposium on Computer Architecture (ISCA), June 2–4, 1997, Denver, CO.

Sohi, G.S., 1990. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. IEEE Trans. Comput. 39 (3), 349–359.

Sohi, G.S., Vajapeyam, S., 1989. Tradeoffs in instruction format design for horizontal architectures. In: Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 3–6, 1989, Boston, pp. 15–25.

Sony/Toshiba, 1999. 'Emotion Engine' in PS2 ("IPU is basically an MPEG2 decoder…"). http://www.cpu-collection.de/?l0=co&l1=Sony&l2=Emotion+Engine http://arstechnica.com/gadgets/2000/02/ee/3/.

Soundararajan, V., Heinrich, M., Verghese, B., Gharachorloo, K., Gupta, A., Hennessy, J.L., 1998. Flexible use of memory for replication/migration in cache-coherent DSM multiprocessors. In: Proceedings of 25th Annual International Symposium on Computer Architecture (ISCA), July 3–14, 1998, Barcelona, Spain, pp. 342–355.

SPEC, 1989. SPEC Benchmark Suite Release 1.0 (October 2).

SPEC, 1994. SPEC Newsletter (June).

Sporer, M., Moss, F.H., Mathais, C.J., 1988. An introduction to the architecture of the Stellar Graphics supercomputer. In: Proceedings of IEEE COMPCON, February 29–March 4, 1988, San Francisco, p. 464.

Spurgeon, C., 2001. Charles Spurgeon's Ethernet Web Site. www.host.ots.utexas.edu/ethernet/ethernet-home.html.

Steinberg, D., 2015. Full-Chip Simulations, Keys to Success. In: Proceedings of the Synopsys Users Group (SNUG) Silicon Valley 2015.

Stenström, P., Joe, T., Gupta, A., 1992. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 80–91.

Sterling, T., 2001. Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux. MIT Press, Cambridge, MA.

Stern, N., 1980. Who invented the first electronic digital computer? Ann. Hist. Comput. 2 (4), 375–376.

Stevens, W.R., 1994–1996. TCP/IP Illustrated (three volumes). Addison-Wesley, Reading, MA.

Stokes, J., 2000. Sound and Vision: A Technical Overview of the Emotion Engine. arstechnica.com/reviews/1q00/playstation2/ee-1.html.

Stone, H., 1991. High Performance Computers. Addison-Wesley, New York.

Strauss, W., 1998. DSP Strategies 2002. www.usadata.com/market_research/spr_05/spr_r127-005.htm.

Strecker, W.D., 1976. Cache memories for the PDP-11? In: Proceedings of Third Annual International Symposium on Computer Architecture (ISCA), January 19–21, 1976, Tampa, FL, pp. 155–158.

Strecker, W.D., 1978. VAX-11/780: a virtual address extension of the PDP-11 family. In: Proceedings of AFIPS National Computer Conference, June 5–8, 1978, Anaheim, CA. vol. 47, pp. 967–980.

Sugumar, R.A., Abraham, S.G., 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. In: Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 17–21, 1993, Santa Clara, CA, pp. 24–35.

Sun Microsystems, 1989. The SPARC Architectural Manual, Version 8, Part No. 8001399-09. Sun Microsystems, Santa Clara, CA.

Sussenguth, E., 1999. IBM's ACS-1 machine. IEEE Comput. 22, 11.

Swan, R.J., Bechtolsheim, A., Lai, K.W., Ousterhout, J.K., 1977a. The implementation of the Cm* multi-microprocessor. In: Proceedings of AFIPS National Computing Conference, June 13–16, 1977, Dallas, TX, pp. 645–654.

Swan, R.J., Fuller, S.H., Siewiorek, D.P., 1977b. Cm*—a modular, multi-microprocessor. In: Proceedings of AFIPS National Computing Conference, June 13–16, 1977, Dallas, TX, pp. 637–644.

Swartzlander, E. (Ed.), 1990. Computer Arithmetic. IEEE Computer Society Press, Los Alamitos, CA.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2015. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

Takagi, N., Yasuura, H., Yajima, S., 1985. High-speed VLSI multiplication algorithm with a redundant binary addition tree. IEEE Trans. Comput. C-34 (9), 789–796.

Talagala, N., 2000. Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks (Ph.D. dissertation). Computer Science Division, University of California, Berkeley.

Talagala, N., Patterson, D., 1999. An Analysis of Error Behavior in a Large Storage System, Tech. Report UCB//CSD-99-1042. Computer Science Division, University of California, Berkeley.

Talagala, N., Arpaci-Dusseau, R., Patterson, D., 2000a. Micro-Benchmark Based Extraction of Local and Global Disk Characteristics, CSD-99-1063. Computer Science Division, University of California, Berkeley.

Talagala, N., Asami, S., Patterson, D., Futernick, R., Hart, D., 2000b. The art of massive storage: a case study of a Web image archive. Computer 33 (11), 22–28.

Tamir, Y., Frazier, G., 1992. Dynamically-allocated multi-queue buffers for VLSI communication switches. IEEE Trans. Comput. 41 (6), 725–734.

Tanenbaum, A.S., 1978. Implications of structured programming for machine architecture. Commun. ACM 21 (3), 237–246.

Tanenbaum, A.S., 1988. Computer Networks, second ed. Prentice Hall, Englewood Cliffs, NJ.

Tang, C.K., 1976. Cache design in the tightly coupled multiprocessor system. In: Proceedings of AFIPS National Computer Conference, June 7–10, 1976, New York, pp. 749–753.

Tanqueray, D., 2002. The Cray X1 and supercomputer road map. In: Proceedings of 13th Daresbury Machine Evaluation Workshop, December 11–12, 2002, Daresbury Laboratories, Daresbury, Cheshire, UK.

Tarjan, D., Thoziyoor, S., Jouppi, N., 2005. HPL Technical Report on CACTI 4.0. www.hpl.hp.com/techeports/2006/HPL=2006+86.html.

Taylor, G.S., 1981. Compatible hardware for division and square root. In: Proceedings of 5th IEEE Symposium on Computer Arithmetic, May 18–19, 1981, University of Michigan, Ann Arbor, MI, pp. 127–134.

Taylor, G.S., 1985. Radix 16 SRT dividers with overlapped quotient selection stages. In: Proceedings of Seventh IEEE Symposium on Computer Arithmetic, June 4–6, 1985, University of Illinois, Urbana, IL, pp. 64–71.

Taylor, G., Hilfinger, P., Larus, J., Patterson, D., Zorn, B., 1986. Evaluation of the SPUR LISP architecture. In: Proceedings of 13th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1986, Tokyo.

Taylor, M.B., Lee, W., Amarasinghe, S.P., Agarwal, A., 2005. Scalar operand networks. IEEE Trans. Parallel Distrib. Syst. 16 (2), 145–162.

Tendler, J.M., Dodson, J.S., Fields Jr., J.S., Le, H., Sinharoy, B., 2002. Power4 system microarchitecture. IBM J. Res. Dev. 46 (1), 5–26.

TensorFlow Tutorials, 2016. https://www.tensorflow.org/versions/r0.12/tutorials/index.html.

Texas Instruments, 2000. History of Innovation: 1980s. www.ti.com/corp/docs/company/history/1980s.shtml.

Tezzaron Semiconductor, 2004. Soft Errors in Electronic Memory, White Paper. Tezzaron Semiconductor, Naperville, IL http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf.

Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., Boggs, D.R., 1982. Alto: a personal computer. In: Siewiorek, D.P., Bell, C.G., Newell, A. (Eds.), Computer Structures: Principles and Examples. McGraw-Hill, New York, pp. 549–572.

Thadhani, A.J., 1981. Interactive user productivity. IBM Syst. J. 20 (4), 407–423.

Thekkath, R., Singh, A.P., Singh, J.P., John, S., Hennessy, J.L., 1997. An evaluation of a commercial CC-NUMA architecture—the CONVEX Exemplar SPP1200. In: Proceedings of 11th International Parallel Processing Symposium (IPPS), April 1–7, 1997, Geneva, Switzerland.

Thorlin, J.F., 1967. Code generation for PIE (parallel instruction execution) computers. In: Proceedings of Spring Joint Computer Conference, April 18–20, 1967, Atlantic City, NJ, p. 27.

Thornton, J.E., 1964. Parallel operation in the Control Data 6600. In: Proceedings of AFIPS Fall Joint Computer Conference, Part II, October 27–29, 1964, San Francisco. 26, pp. 33–40.

Thornton, J.E., 1970. Design of a Computer, the Control Data 6600. Scott Foresman, Glenview, IL.

Tjaden, G.S., Flynn, M.J., 1970. Detection and parallel execution of independent instructions. IEEE Trans. Comput. C-19 (10), 889–895.

Tomasulo, R.M., 1967. An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. 11 (1), 25–33.

Torrellas, J., Gupta, A., Hennessy, J., 1992. Characterizing the caching and synchronization performance of a multiprocessor operating system. In: Proceedings of Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 12–15, 1992, Boston (SIGPLAN Notices 27:9 (September), pp. 162–174.

Touma, W.R., 1993. The Dynamics of the Computer Industry: Modeling the Supply of Workstations and Their Components. Kluwer Academic, Boston.

Tuck, N., Tullsen, D., 2003. Initial observations of the simultaneous multithreading Pentium 4 processor. In: Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03), September 27–October 1, 2003, New Orleans, LA, pp. 26–34.

Tullsen, D.M., Eggers, S.J., Levy, H.M., 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In: Proceedings of 22nd Annual International Symposium on Computer Architecture (ISCA), June 22–24, 1995, Santa Margherita, Italy, pp. 392–403.

Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., 1996. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of 23rd Annual International Symposium on Computer Architecture (ISCA), May 22–24, 1996, Philadelphia, PA, pp. 191–202.

Tung, L., 2016. Google Translate: 'This landmark update is our biggest single leap in 10 years', ZDNet. http://www.zdnet.com/article/google-translate-this-landmark-update-is-our-biggest-single-leap-in-10years/.

Ungar, D., Blau, R., Foley, P., Samples, D., Patterson, D., 1984. Architecture of SOAR: Smalltalk on a RISC. In: Proceedings of 11th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1984, Ann Arbor, MI, pp. 188–197.

Unger, S.H., 1958. A computer oriented towards spatial problems. Proc. Inst. Radio Eng. 46 (10), 1744–1750.

Vahdat, A., Al-Fares, M., Farrington, N., Niranjan Mysore, R., Porter, G., Radhakrishnan, S., 2010. Scale-out networking in the data center. IEEE Micro 30 (4), 29–41.

Vaidya, A.S., Sivasubramaniam, A., Das, C.R., 1997. Performance benefits of virtual channels and adaptive routing: an application-driven study. In: Proceedings of ACM/IEEE Conference on Supercomputing, November 16–21, 1997, San Jose, CA.

Vajapeyam, S., 1991. Instruction-Level Characterization of the Cray Y-MP Processor (Ph.D. thesis). Computer Sciences Department, University of Wisconsin-Madison.

van Eijndhoven, J.T.J., Sijstermans, F.W., Vissers, K.A., Pol, E.J.D., Tromp, M.I.A., Struik, P., Bloks, R.H.J., van der Wolf, P., Pimentel, A.D., Vranken, H.P.E., 1999. Trimedia CPU64 architecture. In: Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'99), October 10–13, 1999, Austin, TX, pp. 586–592.

Van Vleck, T., 2005. The IBM 360/67 and CP/CMS. http://www.multicians.org/thvv/360-67.html.

Vanhoucke, V., Senior, A., Mao, M.Z., 2011. Improving the speed of neural networks on CPUs. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37631.pdf.

von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E., 1992. Active messages: a mechanism for integrated communication and computation. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia.

Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A., 1997. Baring it all to software: raw machines. IEEE Comput. 30, 86–93.

Wakerly, J., 1989. Microcomputer Architecture and Programming. Wiley, New York.

Wall, D.W., 1991. Limits of instruction-level parallelism. In: Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 8–11, 1991, Palo Alto, CA, pp. 248–259.

Wall, D.W., 1993. Limits of Instruction-Level Parallelism, Research Rep. 93/6, Western Research Laboratory. Digital Equipment Corp., Palo Alto, CA.

Walrand, J., 1991. Communication Networks: A First Course. Aksen Associates/Irwin, Homewood, IL.

Wang, W.-H., Baer, J.-L., Levy, H.M., 1989. Organization and performance of a two-level virtual-real cache hierarchy. In: Proceedings of 16th Annual International Symposium on Computer Architecture (ISCA), May 28–June 1, 1989, Jerusalem, pp. 140–148.

Watanabe, T., 1987. Architecture and performance of the NEC supercomputer SX system. Parallel Comput. 5, 247–255.

Waters, F. (Ed.), 1986. IBM RT Personal Computer Technology, SA 23-1057. IBM, Austin, TX.

Watson, W.J., 1972. The TI ASC—a highly modular and flexible super processor architecture. In: Proceedings of AFIPS Fall Joint Computer Conference, December 5–7, 1972, Anaheim, CA, pp. 221–228.

Weaver, D.L., Germond, T., 1994. The SPARC Architectural Manual, Version 9. Prentice Hall, Englewood Cliffs, NJ.

Weicker, R.P., 1984. Dhrystone: a synthetic systems programming benchmark. Commun. ACM 27 (10), 1013–1030.

Weiss, S., Smith, J.E., 1984. Instruction issue logic for pipelined supercomputers. In: Proceedings of 11th Annual International Symposium on Computer Architecture (ISCA), June 5–7, 1984, Ann Arbor, MI, pp. 110–118.

Weiss, S., Smith, J.E., 1987. A study of scalar compilation techniques for pipelined supercomputers. In: Proceedings of Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 5–8, 1987, Palo Alto, CA, pp. 105–109.

Weiss, S., Smith, J.E., 1994. Power and PowerPC. Morgan Kaufmann, San Francisco.

Wendel, D., Kalla, R., Friedrich, J., Kahle, J., Leenstra, J., Lichtenau, C., Sinharoy, B., Starke, W., Zyuban, V., 2010. The Power7 processor SoC. In: Proceedings of International Conference on IC Design and Technology, June 2–4, 2010, Grenoble, France, pp. 71–73.

Weste, N., Eshraghian, K., 1993. Principles of CMOS VLSI Design: A Systems Perspective, 2nd ed. Addison-Wesley, Reading, MA.

Wiecek, C., 1982. A case study of the VAX 11 instruction set usage for compiler execution. In: Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 1–3, 1982, Palo Alto, CA, pp. 177–184.

Wilkes, M., 1965. Slave memories and dynamic storage allocation. IEEE Trans. Electron. Comput. EC-14 (2), 270–271.

Wilkes, M.V., 1982. Hardware support for memory protection: capability implementations. In: Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 1–3, 1982, Palo Alto, CA, pp. 107–116.

Wilkes, M.V., 1985. Memoirs of a Computer Pioneer. MIT Press, Cambridge, MA.

Wilkes, M.V., 1995. Computing Perspectives. Morgan Kaufmann, San Francisco.

Wilkes, M.V., Wheeler, D.J., Gill, S., 1951. The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, Cambridge, MA.

Williams, T.E., Horowitz, M., Alverson, R.L., Yang, T.S., 1987. A self-timed chip for division. In: Losleben, P. (Ed.), 1987 Stanford Conference on Advanced Research in VLSI. MIT Press, Cambridge, MA.

Williams, S., Waterman, A., Patterson, D., 2009. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52 (4), 65–76.

Wilson Jr., A.W., 1987. Hierarchical cache/bus architecture for shared-memory multiprocessors. In: Proceedings of 14th Annual International Symposium on Computer Architecture (ISCA), June 2–5, 1987, Pittsburgh, PA, pp. 244–252.

Wilson, R.P., Lam, M.S., 1995. Efficient context-sensitive pointer analysis for C programs. In: Proceedings of ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, June 18–21, 1995, La Jolla, CA, pp. 1–12.

Wolfe, A., Shen, J.P., 1991. A variable instruction stream extension to the VLIW architecture. In: Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 8–11, 1991, Palo Alto, CA, pp. 2–14.

Wood, D.A., Hill, M.D., 1995. Cost-effective parallel computing. IEEE Comput. 28 (2), 69–72.

Wu, Y., Schuster, M., Chen, Z., Le, Q., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., Dean, J., 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. http://arxiv.org/abs/1609.08144.

Wulf, W., 1981. Compilers and computer architecture. Computer 14 (7), 41–47.

Wulf, W., Bell, C.G., 1972. C.mmp—a multi-mini-processor. In: Proceedings of AFIPS Fall Joint Computer Conference, December 5–7, 1972, Anaheim, CA, pp. 765–777.

Wulf, W., Harbison, S.P., 1978. Reflections in a pool of processors—an experience report on C.mmp/Hydra. In: Proceedings of AFIPS National Computing Conference, June 5–8, 1978, Anaheim, CA, pp. 939–951.

Wulf, W.A., McKee, S.A., 1995. Hitting the memory wall: implications of the obvious. ACM SIGARCH Comput. Architect. News 23 (1), 20–24.

Wulf, W.A., Levin, R., Harbison, S.P., 1981. Hydra/C.mmp: An Experimental Computer System. McGraw-Hill, New York.

Yamamoto, W., Serrano, M.J., Talcott, A.R., Wood, R.C., Nemirosky, M., 1994. Performance estimation of multistreamed, superscalar processors. In: Proceedings of 27th Annual Hawaii International Conference on System Sciences, January 4–7, 1994, Maui, pp. 195–204.

Yang, Y., Mason, G., 1991. Nonblocking broadcast switching networks. IEEE Trans. Comput. 40 (9), 1005–1015.

Yeager, K., 1996. The MIPS R10000 superscalar microprocessor. IEEE Micro 16 (2), 28–40.

Yeh, T., Patt, Y.N., 1993a. Alternative implementations of two-level adaptive branch prediction. In: Proceedings of 19th Annual International Symposium on Computer Architecture (ISCA), May 19–21, 1992, Gold Coast, Australia, pp. 124–134.

Yeh, T., Patt, Y.N., 1993b. A comparison of dynamic branch predictors that use two levels of branch history. In: Proceedings of 20th Annual International Symposium on Computer Architecture (ISCA), May 16–19, 1993, San Diego, CA, pp. 257–266.

Index

Page references in bold represent figures, tables and boxes.

A

- Absolute addressing mode, K-34
- Accelerated Strategic Computing Initiative (ASCI)
  - ASCI Red, F-104–105
  - ASCI White, F-71, F-105
- Access 1/Access 2 stages, TI 320C55 DSP, E-7
- Access bit, B-52
- Access time. *See also* Average memory access time (AMAT)
  - DRAM/magnetic disk, **D-3**
  - DSM, 372–373
  - memory hierarchy design, 85
  - slowdown causes, B-3, **B-3**
  - SMPs, 371
- Access time gap, D-3
- Accumulator, 557–558
  - architecture, A-3
  - extended, A-3
- Acknowledgment, packets, F-17
- ACM. *See* Association for Computing Machinery (ACM)
- ACS project, M-29–30
- Activation hardware, 557–558
- Ada language, integer division/remainder, **J-12**
- Adaptive routing
  - definition, F-47–48
  - *vs.* deterministic routing, **F-53–56**
  - network fault tolerance, F-98
  - and overhead, F-97
- Adders
  - carry-lookahead, J-37–41
  - chip comparison, J-61
  - full, J-2–3, **J-3**
  - half, J-2–3
  - integer division speedup, J-54–57
  - integer multiplication speedup
    - even/odd array, **J-52**
    - many adders, J-50–54, **J-50**
    - multipass array multiplier, **J-51**
    - signed-digit addition table, **J-54**
    - single adder, J-47–49, **J-48–49**
    - Wallace tree, **J-53**
  - radix-2 division, **J-55**
  - radix-4 division, **J-56**
  - radix-4 SRT division, **J-57**
  - ripple-carry, J-3, **J-3**
  - time/space requirements, **J-44**
- Addition operations
  - chip comparison, J-61
  - floating point
    - denormals, J-26–27
    - overview, J-21–25
    - rules, **J-24**
    - speedup, J-25–26
  - integer, speedup
    - carry-lookahead, J-37–41
    - carry-lookahead circuit, **J-38**
    - carry-lookahead tree, **J-40**
    - carry-lookahead tree adder, **J-41**
    - carry-select adder, J-43–44, **J-43–44**
    - carry-skip adder, J-41–43, **J-42**
    - overview, J-37
  - ripply-carry addition, J-2–3, **J-3**
- Address aliasing prediction, 239–240
- Address fault, B-42
- Addressing modes
  - absolute, K-34
  - based indexed addressing, K-34
  - base plus scaled indexed, K-34
  - for control flow instructions, A-17–18
  - data addressing modes, K-32–35
  - displacement, A-11–12
  - and instruction formats, K-6–9
  - instruction set architecture, 13
  - memory addressing, A-8–11, **A-10**
  - register indirect, K-34
  - RISC-V, A-36
- Address offset, B-55–56
- Address space
  - global, B-52
  - local, B-52
  - memory hierarchy, B-57
  - shared memory, 373
  - virtual memory, B-12, B-40, **B-41**, B-44, B-55
- Address specifier, A-21, K-54
- Address stage, TI 320C55 DSP, E-7
- Address trace, B-4
- Address translation, B-42
  - AMD64 paged virtual memory, B-55
  - during indexing, B-36–40, 83
  - Opteron data TLB, **B-47**
  - translation lookaside buffers, B-37, B-46, **B-47**
  - virtual memory, B-46, **B-47**, 120
- Advanced load address table (ALAT)
  - IA-64 ISA, H-40
  - vector sparse matrices, G-12–13
- Advanced loads, IA-64 ISA, H-40

Advanced mobile phone service (AMPS), cell phones, E-25
Advanced Research Project Agency (ARPA), F-102–103
Advanced RISC Machine (ARM), 12
  architecture, K-22
  GPU computing history, M-53
Advanced Simulation and Computing (ASC) program, F-106
Advanced Switching Interconnect (ASI), F-107
Advanced Switching SAN, F-71
Advanced Technology Attachment (ATA) disks
  Berkeley's Tertiary Disk project, D-12–13
  disk power, D-5
  disk storage, D-4
  historical background, M-88
  RAID 6, D-8–9
Advanced vector extensions (AVX), 282, 305, **306**
Affine, loop-level parallelism dependences, H-6
After rounding rule, J-36
Aggregate bandwidth
  definition, F-13
  effective bandwidth calculations, F-18–19
  shared- *vs.* switched-media networks, **F-23**, F-25
  switched-media networks, F-24–25
  switch microarchitecture, F-56
AI. *See* Artificial intelligence (AI)
Aiken, Howard, M-3–4
ALAT. *See* Advanced load address table (ALAT)
Alewife machine, M-62
ALGOL, M-17–18
Aliases, address translation, B-38
Allen, Fran, M-29–30
Alliant processors, G-26
Alloy cache, 115
Alloyed predictors, 184
Alpha 21164
  cache hierarchy, characteristics, **395**
  L1 caches, **395**
AlphaServer, 395–396
AlphaServer 4100, 395
AltaVista search, cluster history, M-63, M-74–75
ALUs. *See* Arithmetic-logical units (ALUs)
AMAT. *See* Average memory access time (AMAT)
Amazon
  Dynamo key-value storage system, 485–486
  Elastic Computer Cloud, 491–492
  Simple Storage Service, 491–492
  warehouse-scale computers, 10
Amazon Elastic Computer Cloud (EC2), utility computing, M-75–76
Amazon Web Services (AWS)
  availability zones, 497–501, **497**
  cloud computing, 491–497
  EC2 computer unit, **493–494**
  growth, **500**
  guarantee of service, 492
  low cost, 492
  reliance on open source software, 492
  virtual machines, 491
  Xen virtual machine, 126
Amdahl, Gene, M-29–30
Amdahl's Law, 5
  computer design principles, 49–52
  computer system power consumption case study, 69–71
  execution time, 50
  multicore scaling, 436, 438, 442
  parallel processing calculations, 373–377
  pitfall, 61
  software overhead, F-96
  speedup, 374–375
  VMIPS on Linpack, G-18
AMD Athlon 64, Itanium 2 comparison, **H-43**
AMD Fusion, M-53
AMD K-5, M-35
AMD Opteron, 387–388
  address translation, B-38
  data cache example, B-12–15, **B-13**
  implementation, 391
  microprocessor, **27**
  misses per instruction, **B-15**
  NetApp FAS6000 filer, D-42
  paged virtual memory example, B-54–57
  vs. Pentium protection, B-57
  processors, 403
  TLB during address translation, **B-47**
  tournament predictors, 185–187
AMD processors
  GPU computing history, M-53
  power consumption, F-89
  recent advances, M-35
  RISC history, M-23
Amortization of overhead, D-64–67
Ample parallelism, 467
Andreessen, Marc, F-102
Andrew benchmark, 399
Annual failure rate, 62
Antenna, radio receiver, **E-23**
Antialiasing, B-38
Antidependence
  compiler history, M-32
  definition, 172
  finding, H-7–8
  register renaming, 196
Apollo DN 10000, M-32
Application layer, **F-84**
Applied Minds, M-76
Arbitration algorithm
  collision detection, F-23–24
  commercial interconnection networks, **F-57**
  interconnection networks, F-21–22, F-49–51
  network impact, F-52–53
  SAN characteristics, F-76–77
  switched-media networks, F-24–25
  switch microarchitecture, F-56
    pipelining, F-65–66
  system area network, F-104–105
Architecturally visible registers, 234
Architectural Support for Compilers and Operating Systems (ASPLOS), M-12
Architecture. *See also* Computer architecture; Instruction set architecture (ISA)
  and compiler writer, A-30–31
  microarchitecture, 266–273
Areal density, D-2
Argument pointer, K-57
Arithmetic intensity, 307–308

Arithmetic-logical units (ALUs)
  data forwarding, **C-36–37**
  data hazards stall minimization, C-14–17
  DSP media extensions, E-10
  effective address cycle, C-5
  IA-64 instructions, **H-35**
  integer division, J-54
  integer multiplication, J-48
  integer operations, C-46–48
  integer shifting over zeros, J-45
  latency, C-46–48
  load interlocks, **C-35**
  micro-op fusion, 254
  MIPS R4000 pipeline, C-59
  multicycle implementation, C-29
  operation, C-27–28
  pipeline branch issues, C-35–36
  RISC classic pipeline, C-8
  RISC instruction set, C-5
  RISC pipeline, C-31–32, C-35
  TX-2, M-50
ARM. *See* Advanced RISC Machine (ARM)
ARM AMBA, OCNs, F-3
ARM Cortex-A53
  characteristics, **259**
  clock cycles per instruction, 251–252, **252**
  data miss rate, **132**
  memory hierarchy design, 129–131, **130**
  misprediction rate, **250**
  multiple-issue processors, 247–252
  pipeline performance, **249**, 250–252
  virtual address, physical and data blocks, **131**
ARMv8, **K-4**, **K-9**, 13, **K-15**, K-16, K-22
ARPANET, F-102
Array
  FFT kernel, I-7
  ocean application, I-9–10
  recurrences, H-12
Array multiplier
  example, **J-50**
  integers, **J-50**
  multipass system, **J-51**
Artificial intelligence (AI), 546
ASC Purple, F-71, F-105
ASPLOS. *See* Architectural Support for Compilers and Operating Systems (ASPLOS)
Assembly language, 2
Association for Computing Machinery (ACM), M-3
Associativity. *See also* Set associativity
  Opteron data cache, B-13–14, **B-13**
  sizes and, **B-10**
Astronautics ZS-1, M-31
Asynchronous events, exception, C-39
Asynchronous I/O, D-35–36
Asynchronous Transfer Mode (ATM)
  interconnection networks, F-102–103
  LAN, F-93–94
  packet format, **F-79**
  total time statistics, **F-94**
  VOQs, F-60–61
  WANs, F-4, F-84–85, F-102–103
ATA disks. *See* Advanced Technology Attachment (ATA) disks
Atanasoff Berry Computer (ABC), M-5
Atanasoff, John, M-5
ATI Radeon 9700, M-51–52
Atlas computer, M-9
ATM system, TP benchmarks, D-18
Atom 230, 258, **259**
Atomic exchange, 413
Atomic instructions, barrier synchronization, **I-14**
Atomic operations, 386
Attributes field, B-52
Autoincrement deferred addressing, K-52–53
Autonet, F-49
Autonomous instruction fetch units, 127
Availability
  computer systems, D-43
  I/O system design/evaluation, D-36–37
Average instruction execution time, M-6
Average memory access time (AMAT)
  block size calculations, B-26–28, **B-28**
  cache optimizations, B-22, B-26–28
  cache performance, B-15–17, **B-22**
  memory hierarchy design, 82
  miss rate, B-29–30, **B-30**
  out-of-order computer, B-21
  and processor performance, B-17–20
  using miss rates, **B-30**
Average reception factor
  centralized switched networks, F-33
  multi-device interconnection networks, F-26–27
AWS. *See* Amazon Web Services (AWS)

B
Back-off time, shared-media networks, F-103
Backpressure, congestion management, F-69
Backpropagation, 548
Backside bus, 377
Balanced systems, D-64–67
Balanced tree, MINs with nonblicking, F-34
Bandwidth. *See also* Cache bandwidth; Throughput
  arbitration, F-49–50
  bisection, F-39–40, F-93, 478
  and cache miss, B-2
  communication mechanism, I-3
  compute, 350
  congestion management, F-68
  Cray Research T3D, F-91
  definition, F-13
  disparity, **F-29**
  FP arithmetic, J-62
  gap, disk storage, D-3
  instruction fetch, 228–232, **229–230**
  latency and effective, F-25–30
  log-log plot, **21**
  memory, 350, 356
  network performance and topology, F-41
  over latency, 20
  point-to-point links and switches, D-34
  shared- *vs.* switched-media networks, **F-23**, F-25
  snooping, 389–390
  two-device networks, F-13–20
  for vector load/store units, 298–299

Banerjee, Uptal, M-32
Bank busy time, vector memory systems, G-9
Banked memory, 346
  vector architectures, **G-10**
Barcelona Supercomputer Center, **F-80**
Barnes
  characteristics, I-8–9
  distributed-memory multiprocessor, **I-32**
  symmetric shared-memory multiprocessors, I-21–22, **I-23**, I-25–26
Barnes-Hut *n*-body algorithm, I-8–9
Barriers
  Cray X1, G-23
  fetch-and-increment, I-20–21
  large-scale multiprocessor synchronization, I-20–21
  large-scale multiprocessor, synchronization, I-13–16, **I-14**, **I-16**, **I-19**, I-20
Based indexed addressing mode, K-34
Base field, B-52
Base plus scaled indexed addressing mode, K-34–35
Base station, E-22–23
Batches, DNNs, 556
Batch processing workloads, 467
Bay Area Research Network (BARRNet), **F-83**
BBN Butterfly, M-61
BBN Monarch, M-61
Before rounding rule, J-36
Benchmarks. *See also* Thread Block; *specific benchmark*
  desktop benchmarks, 41–43
  distribution of data accesses by, **A-14**
  EEMBC, E-12, **E-12**
  embedded applications
    basic considerations, E-12
    power consumption and efficiency, E-13, **E-13–14**
  fallacy, 61
  performance measurement, 40–45
  response time restrictions, **D-18**
  sorting case study, D-64–67
  suite, 41
Beneŝ topology, F-33, **F-34**
Berkeley's Tertiary Disk project
  failures of components, **D-12**
  overview, D-12–13
  system log, **D-43**
Berners-Lee, Tim, F-102
Bertram, Jack, M-29–30
Best-case lower bounds, F-26
Best-case upper bounds, F-26
Between instructions exception, C-39, C-45
Biased exponent, J-15–16
Bidirectional multistage interconnection networks, F-33–34
Bidirectional rings, F-36
Big Endian
  byte order, A-7
  interconnection networks, F-12
BINAC, M-5
Binary code compatibility, embedded systems, E-15
Binary-coded decimal, A-14
Binary-to-decimal conversion, FP precisions, J-33–34
Bing search engine
  negative impact, **486**
  WSCs, 485
Bisection bandwidth, 478
  as network cost constraint, F-93
  network performance and topology, F-93
  NEWS communication, F-42
  topology, F-39
Bisection traffic fraction, F-41–42
Bit error rate (BER), wireless networks, E-21
Bit rot, case study, D-61–64
Bit selection, B-8
Black box network
  basic concept, F-5–6
  effective bandwidth, F-18
  performance, F-13
  switched-media networks, F-24–25
  switched network topologies, F-41
Block. *See also* Cache block
  addressing, B-7–9
  cache optimization, 107–109
  centralized switched networks, F-33
  definition, B-2
  disk array deconstruction, D-51–54
  disk deconstruction case study, D-48–50
  factor, 108
  global code scheduling, H-15–16
  head-of-line, F-59–60
  identification
    memory hierarchy, B-8–9
    virtual memory, B-44–45
  LU kernel, I-8
  memory hierarchy, 81
  multithreading, M-35–36
  network performance and topology, F-41
  offset, B-8–9
    block identification, B-8–9
    cache optimization, B-38
    Opteron data cache, **B-13**, B-14
  placement
    memory hierarchy, B-7–8, **B-7**
    virtual memory, B-44
  RAID performance prediction, D-57–59
  replacement
    memory hierarchy, B-9–10
    virtual memory, B-45
  size, miss rate and, B-26–28, **B-27**
  TI TMS320C55 DSP, E-8
Blocked floating point arithmetic, DSP, E-6
Block servers, *vs.* filers, D-34–35
Block transfer engine (BLT), F-91
Boggs, David, F-103
BOMB, M-4
Booth recoding, J-8–10, **J-9**, J-17
  integer multiplication, **J-49**
Bose-Einstein formula, 34
Bounds checking, B-52
Branch(es)
  completion cycle, C-28
  delayed, C-20, **C-20**
  folding, 231
  history table, C-23
  RISC instruction set, C-5
  VAX, K-57
  WCET, E-4

Branch byte, K-57
Branch hazards, C-18–22
  penalty reduction, C-19–20
  pipeline issues, C-35–37
  scheme performance, C-21–22, **C-22**
  simple scheme examples, C-21–22
Branch penalty
  branch-target buffers, 230–231, **231**
  prediction schemes, **C-21**
  reduction, C-19–20
Branch prediction
  accuracy, **C-25–26**
  buffers, C-23–25, **C-24–26**
  correlation, 182–184
  cost reduction, C-22, 182–191
  dynamic, C-23–25
  early schemes, M-29
  instruction-level parallelism
    correlating branch predictors, 182–184
    Intel Core i7, 190–191
    specialized branch prediction, 232–234
    tagged hybrid predictors, 188–190, **188**, **190**
    tournament predictors, 184–188, **186**
  integrated, 233
  static, C-22, **C-23**
  trace scheduling, H-19
Branch registers, IA-64, H-34
Branch stalls, C-61, **C-64**
Branch-target address
  branch hazards, **C-38**
  pipeline branch issues, C-35–36
  RISC instruction set, C-5
Branch-target buffers
  branch penalty, 230–231, **231**
  handling an instruction with, **230**
  instruction fetch bandwidth, 228–232, **229–230**
  program counter, 228–229, **229**
Branch-target cache, 228
Branch word, K-57
Brewer, Eric, M-74–75
Bridges, **F-82**, F-83
Bubbles, F-47–48, **F-54**
Buckets, D-26
Buffered crossbar switch, F-66
Buffered wormhole switching, F-52
Buffers. *See also* Branch-target buffers
  branch-prediction, C-23–25, **C-24–26**
  DSM multiprocessor cache coherence, I-38–40
  integrated instruction fetch units, 234
  Intel Core i7, **256**
  interconnection networks, F-10–11
  network interface functions, **F-7**
  organizations, F-59–61
  translation lookaside, B-37, B-46, **B-47**
  write, B-11, B-14
Bundles
  example calculations, H-35–36
  IA-64, H-34–35, **H-37**
  Itanium 2, H-41
Burks, Arthur, M-3
Burroughs B5000, M-17–18
Bus-based coherent multiprocessors, M-59–60
Buses
  barrier synchronization, **I-16**
  I/O bus replacements, **D-34**
  large-scale multiprocessor synchronization, I-12–13
  NEWS communication, F-42
  scientific workloads on symmetric shared-memory multiprocessors, **I-25**
  Sony PlayStation 2 Emotion Engine, E-18
  vs. switched networks, F-2
  switch microarchitecture, F-56
Bypassing, C-14
  SAN example, F-77
Byte displacement addressing, K-52
Byte/word/long displacement deferred addressing, K-52–53

C

Cache(s). *See also* Memory hierarchy
  AMD Opteron example, B-12–15, **B-13**, **B-15**
  benefits, 350
  block frames and memory, **B-7**
  concept, M-11
  definition, B-2
  embedded systems, E-4
  Itanium 2, H-42–43
  parameters, **B-42**
  Sony PlayStation 2 Emotion Engine, E-18
  vector processors, G-25
  and virtual memory, B-42–43, **B-42**, B-48–49, **B-48**
Cache bandwidth
  block addressing, **100**
  increasing, 94
  multibanked caches, 99–100
  nonblocking caches, 100–104
  pitfall, 143
Cache block
  cache coherence protocol, 382–383, **384–385**
  definition, B-2
  miss rate reduction, B-26–28
  shared state, 386
  symmetric shared-memory multiprocessors, I-22, I-25–26, **I-25**
  write strategy, B-10–12
Cache coherence
  atomic operations, 386
  of cached data, 128–129
  Cray X1, G-22
  definition, 377–379
  directory-based (*see* Directory-based cache coherence)
  enforcement, 379–380
  example protocol, 383–387, **384**
  extensions, 388
  implementing locks using, 414–417, **416**
  large-scale multiprocessors, I-34–36
    deadlock and buffering, I-38–40
    directory controller, I-40–41
    DSM implementation, I-36–37
  large-scale multiprocessors history, M-61
  mechanism, **384**
  multiprocessor, 377–379, 387
  nonatomic operations, 386
  problem, 377, **378**
  program order, 378–379

Cache coherence *(Continued)*
  snooping (*see* Snooping cache coherence)
  state diagram, **385**, **387**
Cache hit, B-2
  example calculation, B-5
  Opteron data cache, B-14
Cache miss
  block replacement, **B-10**
  definition, B-2
  distributed-memory multiprocessor, **I-32**
  interconnection network, F-91–92
  large-scale multiprocessors, I-34–35
  WCET, E-4
Cache-only memory architecture (COMA), M-61–62
Cache optimization, B-22–40
  advancement, **117**
  cache misses, 112–113
  case study, 148–164
  compiler-controlled prefetching, 111–114
  compiler optimizations, 107–109
  critical word first, 104–105
  early restart, 104–105
  energy consumption, **97**
  fallacy, 142
  floating-point programs, 101–102
  hardware prefetching, 109–111
  HBM packaging, 114–117
  hit time reduction, B-36–40, 95–98
  impact, **B-40**, 148–150
  miss categories, B-23–25
  miss penalty reduction
    read misses *vs.* writes, B-35–36
    via multilevel caches, B-30–35
  miss rate reduction
    via associativity, B-28–30
    via block size, B-26–28, **B-27**
    via cache size, B-28
  multibanked caches, 99–100
  nonblocking caches, 100–104
  pipelined access, 99–100
  power reduction, 95–98
  way prediction, 98–99
  write buffer merging, 105–106, **106**
Cache organization
  block placement, B-7–8, **B-7**
  Opteron data cache, B-12–15, **B-13**
Cache performance, B-3–6, B-15–16
  average memory access time, B-17–20
  cache optimizations impact on, **B-40**
  equations, **B-22**
  example calculation, B-16–17
  miss penalty, B-20–21
  out-of-order execution, B-20–21
Cache prefetch, 111
Cache size, B-13
  miss rate *vs.*, **B-24–25**, B-28, **B-33**, **B-37**
  scientific workloads
    distributed-memory multiprocessors, **I-29–31**
    symmetric shared-memory multiprocessors, I-22–24, **I-24**
  virtually addressed, **B-37**
Caching locks, 415
CACTI
  energy consumption, **97**
  first-level caches, 95–98, **96**
Callee/caller saving, A-19–20
Call gate, **B-53**, B-54
Canonical form, B-55
Capabilities, protection schemes, M-9
Capacity misses
  cache size, **B-24**
  definition, B-23
  memory hierarchy design, 81
  scientific workloads on symmetric shared-memory multiprocessors, I-22, I-24, **I-24**
Capital expenditures (CAPEX), 36, 486–490
Carrier sensing, F-23–24
Carrier signal, wireless networks, E-21
Carry-in, carry-skip adder, J-41–42
Carry-lookahead adder (CLA)
  chip comparison, J-61
  circuit, **J-38**
  early computer arithmetic, J-63
  example calculations, J-39
  integer addition speedup, J-37–41
  with ripple-carry adder, **J-42**
  tree, **J-40–41**
Carry-out
  carry-lookahead circuit, **J-38**
  floating-point addition speedup, J-25
Carry-propagate adder (CPA)
  integer multiplication, J-48, J-51
  multipass array multiplier, **J-51**
Carry-save adder (CSA)
  integer division, J-54–55
  integer multiplication, J-47–48, **J-48**
Carry-select adder
  characteristics, J-43–44
  chip comparison, J-61
  example, **J-43–44**
Carry-skip adder (CSA)
  characteristics, J-41–43
  example, **J-42**, **J-44**
Case statements, A-17
Catapult
  board design, **568**
  CNNs on, 570–572, **571–572**
  evaluating, 601–602
  guidelines, 577–579
  implementation and architecture, 568–569
  search acceleration on, 573–574
  software, 569
  version 1 deployment, 574
  version 2 deployment, 575–577, **576–577**
C/C++ language
  dependence analysis, H-6
  GPU computing history, M-52–53
  integer division/remainder, **J-12**
CDB. *See* Common data bus (CDB)
Cedar project, M-61
Cell, Barnes-Hut *n*-body algorithm, I-9
Cell phones
  block diagram, **E-23**
  embedded system case study
    characteristics, E-22–24
    Nokia circuit board, **E-24**
    overview, E-20
    radio receiver, **E-23**

    - standards and evolution, E-25
    - wireless networks, E-21–22
  - Flash memory, D-3–4
  - Nokia circuit board, **E-24**
  - wireless communication challenges, **E-21**
  - wireless networks, E-21–22
- Centralized shared-memory multiprocessor, 371, 377
  - cache coherence protocol, 377–379, **378**, **384**
    - enforcement, 379–380
    - example protocol, 383–387, **384**
    - extensions, 388
    - state diagram, **385**, **387**
  - invalidate protocol implementation, 382–383
  - local memory, 377
  - SMP and snooping limitations, 389–392
  - snooping coherence protocols, 380, **381**
    - example protocol, 383–387, **384**
    - implementation, 392–393
    - invalidate protocol, **381**
    - limitations, 389–392
    - maintenance, 380–381
  - structure, **372**
- Centralized switched networks, F-31–35, **F-31**
- Centrally buffered switch, F-56
- Central processing unit (CPU)
  - average memory access time, B-18–20
  - DNN and GPUs *vs.*, 595–602
  - early pipelined versions, M-27–28
  - execution time, B-3, B-5, **B-22**
  - GPU computing history, M-53
  - performance measurement history, M-6
  - Sony PlayStation 2 Emotion Engine, E-17–18
  - time, 39
  - TI TMS320C55 DSP, E-8
  - vector memory systems, **G-10**
- Cerf, Vint, F-102
- CFM. *See* Current frame pointer (CFM)
- Chaining
  - convoys, DAXPY code, **G-16**
  - vector processor performance, G-11–12, **G-12**
- Channels, cell phones, E-24
- Charge-coupled device (CCD), Sanyo VPC-SX500 digital camera, E-19
- Checksum
  - dirty bits, D-61–64
  - packet format, **F-7**
- Chime, 291
  - vector chaining, G-11–12
  - vector execution time, G-4
  - vector performance, G-2–4
- Chip-crossing wire delay, F-3, F-74, F-108
- Chip fabrication cost, 67–68
- Chipkill, 94
- Choke packets, F-69–70
- Chunk
  - disk array deconstruction, D-51–54
  - Shear algorithm, D-51–54
- CIFS. *See* Common Internet File System (CIFS)
- Circuit switching, F-51
- Circulating water system (CWS), **483**
- CISC. *See* Complex Instruction Set Computer (CISC)
- CLA. *See* Carry-lookahead adder (CLA)
- Clean block, B-11
- Clock cycle
  - floating-point operations, **C-50**
  - pipeline scheduling, 177–179
  - RISC classic pipeline, C-6
  - RISC exception, C-42–43
  - RISC implementation, **C-30**
  - switch microarchitecture pipelining, F-66
  - vector architectures, G-4
- Clock cycles per instruction (CPI), 53, 559
  - ARM Cortex-A53, 251–252, **252**
  - branch scheme, C-21–22, **C-22**
  - cache behavior impact, B-19–20
  - cache hit calculation, B-5
  - calculation, 375–376
  - clock rate, 261
  - data hazards requiring stalls, C-17
  - instruction-level parallelism, 168–169
  - Intel Core i7 6700, 256, **257**
  - microprocessor advances, M-35
  - pipelined processor, 168–169
  - pipeline with stalls, C-11
  - pipelining concept, C-3
  - processor performance equation, 52
  - RISC history, M-22
  - SPEC92 benchmarks, **C-64**
  - SPECCPUint2006 benchmarks, 256, **257**
  - stalls, **C-64**
- Clock cycle time, 53
  - and associativity, B-29–30
  - cache optimization, B-19–20
  - cache performance, B-3
  - pipeline performance, C-11
  - pipelining, C-3
  - RISC implementation, **C-30**
  - shared- *vs.* switched-media networks, F-25
- Clock rate, 261, **261**
  - microprocessor advances, M-35
- Clock skew, C-8–10
- Clos network, F-33, **F-34**, 510–511, **510–511**
- Cloud computing
  - advantages, 490
  - AWS (*see* Amazon Web Services (AWS))
  - economies of scale, 491
  - fallacy, 514
  - providers, 518
  - utility computing history, M-75–76
- Clusters, 9–10, 369, 478
  - characteristics, **I-45**
  - containers, M-76
  - Cray X1, G-22
  - history background, M-62–65
  - IBM Blue Gene/L, I-41–44, **I-43–44**
  - interconnection network domains, F-3
  - large-scale multiprocessors, I-6
  - large-scale multiprocessor trends, M-63–64
  - power consumption, F-89
  - utility computing, M-75–76
  - as WSC forerunners, M-74–75
- Cm*, M-57

C.mmp, M-57
CMOS
  cell phone, E-24
  first vector computers, M-48
  ripply-carry addition, J-3
  scaling, 442–443
  vector processors, G-25–27
CNN. *See* Convolutional neural network (CNN)
Coarse-grained multithreading
  definition, 243–244
  superscalar processor, 245
Cocke, John, M-20, M-29–31
Code division multiple access (CDMA), cell phones, E-25
Code scheduling
  example, **H-16**
  parallelism, H-15–23
  superblock scheduling, H-21–23, **H-22**
  trace scheduling, H-19–21, **H-20**
Coefficient of variance, D-27
Coerced exception, C-39
Coherence. *See* Cache coherence
Coherence misses, I-22, 82, 393
Cold aisles, 506, **507**
Cold-start misses, B-23
Collision detection, shared-media networks, F-23–24
Collision misses, B-23
Collision, shared-media networks, F-23–24
Collocation sites, interconnection networks, F-89
COLOSSUS, M-4
Column access strobe (CAS), 85–86
Column major order, 107
COMA. *See* Cache-only memory architecture (COMA)
Combining tree, large-scale multiprocessor synchronization, I-18
Command queue depth, *vs.* disk throughput, **D-4**
Commercial interconnection networks
  congestion management, F-68–70
  connectivity, F-67
  cross-company interoperability, F-67–68
  DECstation 5000 reboots, **F-73**
  fault tolerance, F-70–72
Commit stage, 211
Commoditization, and cost, 30–31
Commodity cluster, characteristics, **I-45**
Common data bus (CDB), 197, 201
  performance, 207
  reservation stations and register tags, **202**
  write result, 211
Common Internet File System (CIFS), D-35
  NetApp FAS6000 filer, D-41–42
Communication bandwidth, I-3
Communication latency, I-3
  hiding, I-4
Communication mechanism, 375–376
  internetworking, F-85–89
  large-scale multiprocessors
    advantages, I-4–6
    metrics, I-3–4
  network interfaces, F-7–8
  NEWS communication, **F-42–44**
Communication protocol, F-8
Compare-select-store unit (CSSU), TI TMS320C55 DSP, E-8
Compiler(s)
  constants, A-31
  definition, C-65
  interaction, A-27–30
  for multimedia instructions, A-31–32
  phase, 399
  primitives, A-30
  regularity, A-30
  role, A-24–33
  structure, A-25–26
  trade-offs, A-30
  writer, A-30–31
Compiler-controlled prefetching, 111–114
Compiler optimization
  for caches, 107–109, 148–164
  instruction-level parallelism, 176–182
  memory consistency, 422
Compiler scheduling, hardware support, M-32–33
Compiler speculation, hardware support
  example calculations, H-29
  memory references, H-32
  overview, H-27
  preserving exception behavior, H-28–32
Compiler techniques
  Cray X1, G-21–22
  dependence analysis, H-7–8
  global code scheduling, H-17–18
  vectorization, G-12–14
  vector sparse matrices, G-12
Complex Instruction Set Computer (CISC), K-51
  RISC history, M-23
Compulsory misses, B-23
  cache size, **B-24**
  memory hierarchy design, 81
Computation-to-communication ratios
  parallel programs, I-10–12
  scaling, **I-11**
Compute bandwidth, 350
Compute-optimized processors, F-92
Computer architecture
  definition, 11–12, M-18–19
  floating-point addition, rule, **J-24**
  functional requirements, 17–18, **18**
  goals, 17–18
  high-level language, M-19–20
  instruction set architecture, 12–17
  limits of energy, 28–29
  of warehouse-scale computers, 477–482
Computer arithmetic
  chip comparison, J-57–61, **J-58–60**
  floating point
    denormals, J-14–15
    exceptions, J-34–35
    fused multiply-add, J-32–33
    IEEE 754, **J-16**
    iterative division, J-27–31
    and memory bandwidth, J-62
    number representation, J-15–16
    overview, J-13–14
    precisions, J-33–34
    remainder, J-31–32
    special values, J-14–15, **J-16**
    underflow, J-36–37, J-62
  floating-point addition
    denormals, J-26–27
    overview, J-21–25

  - rules, **J-24**
  - speedup, J-25–26
- floating-point multiplication
  - denormals, J-20–21
  - examples, **J-19**
  - overview, J-17–20
  - rounding, **J-18**, J-19
- integer addition speedup
  - carry-lookahead, J-37–41
  - carry-lookahead circuit, **J-38**
  - carry-lookahead tree, **J-40**
  - carry-lookahead tree adder, **J-41**
  - carry-select adder, J-43–44, **J-43–44**
  - carry-skip adder, J-41–43, **J-42**
  - overview, J-37
- integer arithmetic
  - language comparison, **J-12**
  - overflow, **J-11**
  - Radix-2 multiplication/division, J-4–7, **J-4**
  - restoring/nonrestoring division, J-5, **J-6**
  - ripply-carry addition, J-2–3
  - signed numbers, J-7–10
  - systems issues, J-10–13
- integer division
  - radix-2 division, **J-55**
  - radix-4 division, **J-56**
  - radix-4 SRT division, **J-57**
  - with single adder, J-54–57
  - SRT division, J-45–47, **J-46**, J-55–57
- integer-FP conversions, J-62
- integer multiplication
  - array multiplier, **J-50**
  - Booth recoding, **J-49**
  - even/odd array, **J-52**
  - many adders, J-50–54, **J-50**
  - multipass array multiplier, **J-51**
  - signed-digit addition table, **J-54**
  - with single adder, J-47–49, **J-48–49**
  - Wallace tree, **J-53**
- integer multiplication/division, shifting over zeros, J-45
- overview, J-2
- rounding modes, J-14, J-17–20, **J-18**, **J-20**

Computer chip fabrication, Cray X1E, G-24
Computer classes
  - clusters, 9–10
  - desktop computing, 8
  - embedded computers, 6–7
  - internet of things, 6–7
  - parallel architectures, 10–11
  - parallelism, 10–11
  - personal mobile device, 7–8
  - servers, 8–9
  - and system characteristics, **E-4**
  - warehouse-scale computers, 9–10

Computer clusters, 470
Computer design principles
  - Amdahl's law, 49–52
  - common case, 49
  - locality, 48–49
  - parallelism, 48
  - processor performance equation, 52–55

Computer room air-conditioning (CRAC), 482
Computer technology, improvements, 2–6
Compute tiles, OCNs, F-3
Compute Unified Device Architecture (CUDA), 311
  - CUDA Thread, 311
    - GPU programming, 320
    - SIMD instructions, 325–326
  - GPU computing history, M-52–53

Computing efficiently, at low utilization, 468
Conditional branching
  - global code scheduling, H-16, **H-16**
  - graphics processing units, 323–326
  - options, A-18–19, **A-19**
  - static branch prediction, C-22

Conditional instructions
  - example calculations, H-23–24
  - exposing parallelism, H-23–27
  - limitations, H-26–27

Condition codes, K-11, C-44, K-57
Conflict misses
  - cache optimizations, B-23
  - cache size, **B-24**
  - memory hierarchy design, 82

Congestion control, F-68, F-70
Congestion management, F-68–70
Connectedness, F-30, **F-48**
Connection Machine CM-5, F-96
Connection Multiprocessor 2, M-46, M-56
Consistency. *See* Memory consistency
Constellation, characteristics, **I-45**
Containers, cluster history, M-76
Contention delay, F-26
Context switching, B-49, 119
Control bits, messages, F-7
Control Data Corporation (CDC)
  - CDC 6600, C-66–67
    - computer architecture definition, M-19
    - early computer arithmetic, J-64–65
    - first dynamic scheduling, M-28–29
    - multiple-issue processor development, M-28–29
    - multithreading history, M-35
    - RISC history, M-28–29
  - first vector computers, M-47
  - STAR-100, first vector computers, M-47
  - STAR processor, G-26

Control dependences
  - conditional instructions, H-24
  - global code scheduling, H-16
  - hardware-based speculation, 208
  - instruction-level parallelism, 174–176
  - maintenance, 175

Control flow instructions, 14
  - addressing modes for, A-17–18
  - classes, **A-17**
  - compilers
    - role, A-24–33
    - structure, A-25–26, **A-25**
  - conditional branch options, A-18–19, **A-19**
  - conditional instructions, H-27
  - procedure invocation options, A-19–20
  - RISC-V, A-39–40
  - types, A-16

Control hazards, C-11
Controllers, historical background, M-88–89
Convex Exemplar, M-62
Convex processors, G-26

Convolutional neural network (CNN), 550–552, **551**
  on Catapult, 570–572, **571**
  processing element of, **572**
Convolution, DSP, E-5
Convoy, 290–292
  chained, DAXPY code, **G-16**
  DAXPY on VMIPS, G-20–21
  strip-mined loop, G-5
  vector starting times, G-4
Conway, Lynn, M-29–30
Cooling systems, **483**
Cooling towers, 508
Copper wiring
  Ethernet, F-82
  interconnection networks, F-9–10
Copy propagation, H-10
Core, 17
Core i7, 346–353
Core plus ASIC, embedded systems, E-3
Cortex-A53
  ARM, 129–131, **130**
  performance, 132
Cosmic Cube, M-60–61
Cost
  branch prediction, C-22
  disk storage, D-2
  DRAM/magnetic disk, **D-3**
  interconnecting node calculations, **F-32–33**
  Internet Archive Cluster, D-38–40
  I/O system design/evaluation, D-36
  magnetic storage history, M-85–86
  SIMD supercomputer development, M-45
Cost-performance, 467
  DSAs, 600–601, **601**
  extensive pipelining, C-70
  IBM eServer p5 multiprocessor, 440, **441**
  sorting case study, D-64–67
Cost trends
  cost-sensitive designs, 29
  integrated circuit, 31–35
  manufacturing *vs.* operation, 36
  vs. price, 35
  time, volume, and commoditization, 30–31
Counter register, K-25
CPA. *See* Carry-propagate adder (CPA)
CPI. *See* Clock cycles per instruction (CPI)
CP-67 program, M-10
CPU. *See* Central processing unit (CPU)
Cray-1
  first vector computers, M-47–48
  pipeline depths, G-4
  RISC history, M-20
  vector performance measures, G-16
Cray-2
  DRAM, G-25
  first vector computers, M-47–48
  tailgating, G-20–21
Cray-3, G-27
Cray-4, G-27
Cray C90
  first vector computers, M-48
  vector performance calculations, G-8
Cray J90, M-48
Cray Research programmers, 303
Cray Research T3D, F-91
Cray, Seymour, G-25, G-27, M-47–48
Cray supercomputers, early computer arithmetic, J-63–64
Cray T90, 299
Cray T3D, M-61, F-104–105
Cray T3E, M-49, M-61, F-71, F-98–99
Cray X, 370
Cray X1, M-64
  cluster history, M-64
  first vector computers, M-49
  MSP module, **G-22**, G-23–24
  overview, G-21–23
Cray X1E, G-24, F-91, F-95
Cray X2, first vector computers, M-49
Cray X-MP, M-47–48
Cray XT3, M-59, M-63
Cray XT3 SeaStar, F-67
Cray Y-MP
  first vector computers, M-48
  parallel processing debates, M-58
Create vector index instruction (CVI), G-13
Credit-based control flow, F-10, F-18, F-69, F-75
CRISP, M-29
Critical path
  global code scheduling, H-16
  trace scheduling, H-19–21, **H-20**
Critical word first, cache optimization, 104–105
Crossbars, F-31, **F-31**
  Convex Exemplar, M-62
Crossbar switch
  centralized switched networks, F-31
  interconnecting node calculations, **F-32–33**
Cross-company interoperability, F-67–68
Crusoe, M-32–33
Cryptanalysis, M-4
CSA. *See* Carry-save adder (CSA)
CUDA. *See* Compute Unified Device Architecture (CUDA)
Current frame pointer (CFM), H-33–34
Custom cluster
  characteristics, **I-45**
  IBM Blue Gene/L, I-41–44, **I-43–44**
Cut-through packet switching, F-51–53
CVI. *See* Create vector index instruction (CVI)
CYBER 205, M-47
  vector processor history, G-26–27
Cycle time. *See also* Clock cycle time
  CPI calculation, 375–376
  memory hierarchy design, 85
Cyclic redundancy check (CRC)
  IBM Blue Gene/L 3D torus network, F-76
  network interface, F-8
Cydrome, M-31–33

D

DAG. *See* Directed acyclic graph (DAG)
Dark silicon, 28
DASH multiprocessor, M-61
Data addressing modes, K-32–35
Data cache
  cache performance, B-16–17
  TLB, B-46
Data cache miss
  applications *vs.* OS, **B-59**
  Opteron, B-12–15, **B-13**

  - sizes and associativities, **B-10**
  - writes, **B-10**
- Datacenters, containers, M-76
- Data dependences
  - conditional instructions, H-24
  - definition, 170–172
  - example calculations, H-3–4
  - instruction-level parallelism, 170–172
  - maintenance, 175
- Data fetch (DF), C-58–59
- Data flow
  - control dependence, 174–176
  - execution, hardware-based speculation, 209
  - global code scheduling, H-17–18
  - limit, M-34
- Data hazards
  - definition, C-11
  - dynamic scheduling, 191–201
  - instruction set complications, C-45
  - microarchitectural techniques case study, 266–273
  - pipelined execution of instructions, C-13
  - program order, 173–174
  - stall minimization by forwarding, C-14–15, **C-15–16**
  - stall requirements, C-16–17
  - types, C-12
- Data integration (DI), 44
- Data-level parallelism (DLP), 5, 10–11
  - computer design principles, 48
  - cross-cutting issues
    - banked memory and graphics memory, 346
    - energy and DLP, 345
    - strided accesses and TLB misses, 346
  - energy and, 345
  - graphics processing units
    - conditional branching, 323–326
    - multimedia SIMD computers *vs.*, 335
    - NVIDIA computational structures, 313–320
    - NVIDIA GPU instruction set architecture, 320–323
    - NVIDIA GPU memory structures, 326–328, **327**
    - Pascal GPU architecture, 328–331
    - programming, 310–313
    - quick guide, **314**
    - vector architectures *vs.*, 331–334, **332**
  - loop-level parallelism
    - analysis, 337–339
    - CUDA/NVIDIA term, **337–338**
    - dependent computations, 344–345
    - finding dependences, 339–344
  - SIMD Multimedia Extensions, 304–310
  - vector architecture, 282
    - execution time, 290–293
    - *vs.* graphics processing units, 331–334
    - memory banks, 298–299
    - multidimensional arrays, 299–301
    - multiple lanes, 293–294
    - predicate registers, 296–298
    - processor example, 288–290
    - programming, 302–304
    - RV64V extension, 283–287, **284**
    - sparse matrices, 301–302
    - vector-length registers, 294–296
  - WSCs, 467
- Data link layer
  - definition, **F-84**
  - interconnection networks, F-10
- Data parallelism, M-56
- Data-race-free, 419
- Data races, 419
- Data transfers
  - cache miss rate calculations, B-16
  - RISC-V, A-36
- Data trunks, C-70
- Data types, dependence analysis, H-10
- Dauber, Phil, M-29–30
- DAXPY loop
  - chained convoys, **G-16**
  - on enhanced VMIPS, G-19–21
  - vector performance measures, G-16
  - VMIPS, G-19–21
    - calculations, G-18
    - on Linpack, G-18
    - peak performance, G-17
- D-cache, way prediction, 98–99
- DDR. *See* Double data rate (DDR)
- DDR3 memory systems, 153–155
- Deadlock, F-45–46, 386
  - avoidance, F-46
  - large-scale multiprocessor cache coherence, I-34–35, I-38–40
  - recovery, F-46
- Dead time, vector pipeline, G-8, **G-8**
- DEC Alpha processor, K-3
- Decoder, radio receiver, **E-23**
- Decode stage, TI 320C55 DSP, E-7
- DEC PDP-11, address space, B-57–58
- DECstation 5000, **F-73**
- DEC VAX
  - address space, B-57–58
  - cluster history, M-62, M-74
  - computer architecture definition, M-19
  - early computer arithmetic, J-63–64
  - early pipelined CPUs, M-28
  - failures, D-13–15
  - integer overflow, **J-11**
  - RISC history, M-20
- DEC VAX-11/780, M-6–7, M-11, M-19
- DEC VAX 8700
  - *vs.* MIPS M2000, **M-22**
  - RISC history, M-22
- Dedicated link network
  - black box network, F-5–6
  - effective bandwidth, F-18
  - example, **F-6**
- Deep neural networks (DNNs)
  - acceleration, 606–613
  - activation, 546
  - applications, **547**, **595**
  - batches, 556
  - convolutional neural network, 550–552
  - CPUs and GPUs *vs.*, 595–602
  - multilayer perceptron, 549–550
  - neurons of, 546–547
  - performance summary, 603
  - quantization, 556
  - recurrent neural network, 553–555
  - training set sizes/time, **548**

- Deep neural networks (DNNs) *(Continued)*
  - training *vs.* inference, 547–549
  - weights/parameters, 546
- Defect tolerance, 67–68
- Delayed branch
  - behavior, **C-20**
  - compiler history, M-33
  - definition, C-20
- Dell PowerEdge servers, 55–58, **56**
- Dell Poweredge Thunderbird, **F-80**
- Demand access, memory hierarchy design, 138
- Demodulator, radio receiver, **E-23**
- Dennard scaling, 4–5, 58, 368–369, 442
- Denormals, J-14–15, J-20–21
  - floating point addition, J-26–27
  - floating-point underflow, J-36
- Dense matrix multiplication, LU kernel, I-8
- Density-optimized processors, *vs.* SPEC-optimized, F-89
- Dependability
  - benchmark examples, D-21–23
  - definition, D-10–11
  - disk operators, D-13–15
  - integrated circuits, 36–38
  - Internet Archive Cluster, D-38–40
  - in memory systems, 93–94
  - via redundancy, 467
- Dependence analysis
  - basic approach, H-5
  - example calculations, H-7
  - limitations, H-8–9
- Dependence distance, loop-carried dependences, H-6
- Dependences
  - control, 174–176
  - data, 170–172
  - finding, H-6–10
  - loop-level parallelism, H-3
  - name, 172–173
  - sparse matrices, G-12–13
  - types, 170–171
- Dependent computations, H-10–12, 344–345
- Descriptor privilege level (DPL), **B-53**
- Descriptor tables, B-52
- Design faults, D-11
- Desktop benchmarks, 41–43
- Desktop computers
  - interconnection networks, **F-72**
  - multimedia support, **E-11**
  - RAID history, M-87
  - RISC architectures survey for, K-3–29
  - system characteristics, **E-4**
- Desktop computing, A-2, 8
- Desktop/server RISC architectures, instruction formats for, **K-8**
- Destination offset, IA-32 segment, **B-53**
- Deterministic routing algorithm, F-46–47
- DF. *See* Data fetch (DF)
- Dies, 31
  - embedded systems, E-15
  - Intel Core i7 microprocessor, **32**
  - RISC-V, **33**
  - yield, 33–34
- Digital Alpha
  - conditional instructions, H-27
  - Digital Alpha 21064, M-48
  - processors
    - MAX, multimedia support, **E-11**
    - recent advances, M-35
  - synchronization history, M-64–65
- Digital Equipment Vax, 2
- Digital Linear Tape, M-85
- Digital signal processor (DSP)
  - cell phones, E-23–24, **E-23**
  - definition, E-3
  - desktop multimedia support, **E-11**
  - embedded RISCs, K-28
  - examples and characteristics, **E-6**
  - media extensions, E-10–11
  - overview, E-5–7
  - TI TMS320C55, **E-6–7**, E-7–8
  - TI TMS320C6x, E-8–10
  - TI TMS320C64x, **E-9**
  - TI TMS320C6x instruction packet, **E-10**
- Dimension-order routing (DOR), F-46–47
- Direct attached disks, D-35
- Directed acyclic graph (DAG), 582
- Direct-mapped cache, B-7, **B-8**
  - address translation, B-38
  - early work, M-11
  - memory hierarchy, **B-48**, 81
- Direct memory access (DMA)
  - historical background, M-89
  - network interface functions, F-7
  - Sanyo VPC-SX500 digital camera, E-19
  - Sony PlayStation 2 Emotion Engine, E-18
  - TI TMS320C55 DSP, E-8
  - zero-copy protocols, F-95
- Direct networks, F-35, **F-37**, F-96
- Directory-based cache coherence, 380, 391–392
  - case study, 451–452
  - home node, 406
  - large-scale multiprocessors history, M-61
  - local node, 406
  - operations, 406
  - protocol example, 408–412
  - remote node, 406–407
  - state transition diagram, 408, **409–410**
- Directory-based multiprocessor
  - characteristics, **I-31**
  - scientific workloads, I-26, I-29
  - synchronization, I-16, I-19–20
- Directory controller, cache coherence, I-40–41
- Directory protocol, 404
- DirectX 9, M-51–52
- DirectX 10 generation, M-52–53
- Dirty bit, B-11, B-46, D-61–64
- Dirty block, B-11, B-36
- Discrete cosine transform, DSP, E-5
- Disk arrays
  - deconstruction case study, D-51–54
  - RAID 6, D-8–9
  - RAID levels, D-6–10
- Disk layout, RAID performance prediction, D-57–59
- Disk power, D-5
- Disk storage, D-2–10, D-48–50
- Disk system
  - performance milestones, **22**
  - subsystem, failure rates of, 51–52
  - workload measurements, 400
- Dispatch stage, 266–273
- Displacement addressing, K-52
- Displacement-style addressing mode, A-11–12

- Display lists, Sony PlayStation 2 Emotion Engine, E-17
- Distributed routing, F-49
- Distributed shared memory (DSM), 371, 373
  - access time, 372–373
  - architecture, **373**
  - characteristics, **I-45**
  - directory-based cache coherence, 404–412, **405**
  - disadvantages, 372–373
  - multicore processor, **373**, **405**, **452**
- Distributed shared-memory multiprocessors
  - cache coherence implementation, I-36–37
  - scientific application performance, I-26–32, **I-28–32**
- Distributed switched networks, F-35–40
- Divide operations
  - chip comparison, J-61
  - floating-point iterative, J-27–31
  - integer shifting over zeros, J-45
  - integers, speedup
    - radix-2 division, **J-55**
    - radix-4 division, **J-56**
    - radix-4 SRT division, **J-57**
    - with single adder, J-54–57
    - SRT division, J-45–47, **J-46**, J-55–57
  - language comparison, **J-12**
  - *n*-bit unsigned integers, **J-4**
  - Radix-2, J-4–7, **J-4**
  - restoring/nonrestoring, J-5, **J-6**
  - SRT division, J-45–47, **J-46**
- DLP. *See* Data-level parallelism (DLP)
- DLX, integer arithmetic, J-11–12
- DNNs. *See* Deep neural networks (DNNs)
- Domain-specific architectures (DSAs), 5
  - architecture renaissance, 605–606
  - cost-performance, 600–601
  - CPUs and GPUs *vs.* DNN accelerators, 595–602
  - custom chip, 602
  - deep neural networks
    - activation, 546
    - applications, **547**
    - batches, 556
    - convolutional neural network, 550–552
    - multilayer perceptron, 549–550
    - neurons of, 546–547
    - quantization, 556
    - recurrent neural network, 553–555
    - training set sizes/time, **548**
    - training *vs.* inference, 547–549
    - weights/parameters, 546
  - designing, 604
  - guidelines for, 543–544, **543**
  - heterogeneity, 592–594
  - Intel Crest, 579
  - ISPs, 580–582
  - Microsoft Catapult
    - board design, **568**
    - CNNs on, 570–572, **571–572**
    - evaluating, 601–602
    - guidelines, 577–579
    - implementation and architecture, 568–569
    - search acceleration on, 573–574
    - software, 569
    - version 1 deployment, 574
    - version 2 deployment, 575–577, **576–577**
  - open instruction set, 594
  - performance counters, 603
  - performance/watt, 600–601
  - Pixel Visual Core
    - architecture philosophy, 583–584
    - evaluating, 601–602
    - example, 588
    - floor plan, **592**
    - Halo, 584–585
    - implementation, 590–591
    - instruction set architecture, 587–588
    - line buffers in, 590
    - processing element, 588–589
    - processor, 585–587
    - software, 582
    - two-dimensional array, **586**
    - two-dimensional line buffers, 589–590
  - response time, 596–600
  - rooflines, 596–600
  - system on a chip, 592–594
  - systolic array, **561**
  - TCO, 600–601
  - tensor processing unit
    - architecture, 557–558
    - block diagram, **558**
    - case study, 606–617
    - die, **562**
    - guidelines, 566–567
    - implementation, 560–563
    - improving, 564–566
    - instruction set architecture, 559
    - microarchitecture, 559–560
    - origin, 557
    - printed circuit board, **563**
    - software, 563
    - TensorFlow program, **564**
  - throughput, 596–600
- Double data rate (DDR), 87, 399
  - IBM Blue Gene/L, I-43
  - InfiniBand, F-81
- Double-extended floating-point arithmetic, J-33–34
- Double failures, RAID reconstruction, D-55–57
- Double-precision floating point, **C-63**, 329
  - chip comparison, **J-58**
  - DSP media extensions, E-10
- Double rounding
  - FP precisions, J-34
  - FP underflow, J-36–37
- Double words, A-7, **A-8**, **A-14**, K-35, 300
- DPL. *See* Descriptor privilege level (DPL)
- DRAM. *See* Dynamic random-access memory (DRAM)
- DRDRAM, Sony PlayStation 2, E-16
- Driver domains, Xen virtual machine, 126
- DSAs. *See* Domain-specific architectures (DSAs)
- DSM. *See* Distributed shared memory (DSM)
- DSP. *See* Digital signal processor (DSP)
- Dual inline memory modules (DIMMs), F-74, 89
- Dynamically allocatable multi-queues (DAMQs), F-56
- Dynamically shared libraries, A-18

Dynamic branch prediction, C-23–25
Dynamic energy, 25
Dynamic network reconfiguration, F-71–73
Dynamic power, 80
Dynamic programming feature (DPF), 577
Dynamic random-access memory (DRAM)
  arithmetic operations and energy cost, **29**
  clock rates, bandwidth, and names, **89**
  cost *vs.* access time, **D-3**
  Cray X1, G-22
  dependability, 516
  die stacking, **91**
  disk storage, D-3
  embedded benchmarks, E-13
  errors and faults, D-11
  first vector computers, M-47–49
  IBM Blue Gene/L, I-43–44
  internal organization, **86**
  magnetic storage history, M-86
  memory hierarchy design, 85–87
  memory performance improvement, 87–90
  PlayStation 2, **E-16**, E-17
  price pressures, 34
  semiconductor, 19
  stacked/embedded, 91
  timing parameters, 153–155
  vector memory systems, G-9–10
  vector processor, G-9–10, G-25
Dynamic register typing, 287
Dynamic scheduling
  advantages, 191–192
  data hazards, 191–201
  definition, C-65–66
  first use, M-28–29
  out-of-order execution, 193–194
  with scoreboard, C-66–70, **C-68**
  Tomasulo's algorithm, 195–204
    loop-based example, 204–208, **206**
    steps, **205**
  unoptimized code, C-70
Dynamic voltage-frequency scaling (DVFS), 27

E

Early restart, cache optimization, 104–105
Earth Simulator, M-48–49, M-63
ECC. *See* Error-Correcting Code (ECC)
Eckert, J. Presper, M-2–5, M-20
Eckert-Mauchly Computer Corporation, M-5, M-57
ECL minicomputer, M-20
EEMBC. *See* Electronic Design News Embedded Microprocessor Benchmark Consortium (EEMBC)
Effective address, A-8–9
  RISC classic pipeline, C-8
  RISC instruction set, C-5
  simple RISC implementation, C-27
  TLB, B-49
Effective bandwidth
  definition, F-13
  example calculations, F-18–19
  vs. packet size, **F-19**
  two-device networks, F-13–20
Efficiency factor, F-53, F-55–56
Eight-way set associativity
  cache optimization, B-28–29
  conflict misses, B-23
  data cache misses, **B-10**
Elapsed time, 39
Electronically erasable programmable read-only memory (EEPROM), 92
Electronic Design News Embedded Microprocessor Benchmark Consortium (EEMBC), 41
  benchmark classes, E-12
  kernel suites, **E-12**
  power consumption and efficiency metrics, E-13, **E-13–14**
Electronic Discrete Variable Automatic Computer (EDVAC), M-2–3
Electronic Numerical Integrator and Calculator (ENIAC), M-2–3, M-85
Embedded applications, A-2
Embedded computer, 6–7
  RISC architectures survey for, K-3–29
Embedded DRAM, 91
Embedded multiprocessors, characteristics, E-14–15
Embedded systems
  benchmarks
    basic considerations, E-12
    power consumption and efficiency, E-13, **E-13–14**
  cell phone case study
    block diagram, **E-23**
    characteristics, E-22–24
    Nokia circuit board, **E-24**
    overview, E-20
    radio receiver, **E-23**
    standards and evolution, E-25
    wireless networks, E-21–22
  characteristics, **E-4**
  digital signal processor
    cell phones, E-23–24, **E-23**
    definition, E-3
    desktop multimedia support, **E-11**
    examples and characteristics, **E-6**
    media extensions, E-10–11
    overview, E-5–7
    TI TMS320C55, **E-6–7**, E-7–8
    TI TMS320C6x, E-8–10
    TI TMS320C64x, **E-9**
    TI TMS320C6x instruction packet, **E-10**
  EEMBC benchmark suite, **E-12**
  overview, E-2
  performance, **E-13–14**
  real-time processing, E-3–4
  Sanyo digital cameras, SOC, **E-20**
  Sanyo VPC-SX500 digital camera case study, E-19
  Sony PlayStation 2 case study, E-15–18
    block diagram, **E-16**
    organization, **E-18**
EMC, M-87–88
Emotion Engine
  organization modes, **E-18**
  Sony PlayStation 2 case study, E-15–18
empowerTel Networks, MXP processor, E-14–15

Enclaves, instruction set extensions, 125
Encore Multimax, M-59–60
End-to-end flow control, F-69
Energy
  and DLP, 345
  limits of, 28–29
  within microprocessor, 25–28
  proportionality, 503
  systems perspective, 23–24
Energy efficiency, 434–437, 467. *See also* Power consumption
  embedded benchmarks, E-13
Engineering Research Associates (ERA), M-4–5
ENIAC. *See* Electronic Numerical Integrator and Calculator (ENIAC)
Enigma coding machine, M-4
Entry time, transactions, D-16, **D-17**
Environmental faults, storage systems, D-11
EPIC approach
  historical background, M-33
  IA-64, H-33
E-24 RF. *See* Register fetch (RF)
Error correcting codes (ECCs), 93–94
  disk storage, D-11
  hardware dependability, D-15
  RAID 2, D-6
Error handling, interconnection networks, F-9–12
Errors, definition, D-10
Escape resource set, F-47–48
ETA processor, G-26–27
Ethernet, 478
  and bandwidth, **F-82**, F-93
  LANs, F-4, F-82–84, F-103–104
  packet format, **F-79**
  shared-media networks, F-23–24
  shared- *vs.* switched-media networks, **F-23**
  switch *vs.* NIC, **F-90**
  system area networks, F-76–77
  total time statistics, **F-94**
  WAN, F-84–85
Eugene, Miya, M-65
European Center for Particle Research (CERN), F-102
Even/odd array
  example, **J-52**
  integer multiplication, J-51–52
EVEN-ODD scheme development, D-10
Exception
  arithmetic-logical units, C-5
  categories, **C-40**
  control dependence, 174–175
  floating-point, C-41–42
  floating-point arithmetic, J-34–35
  imprecise, 194
  memory protection, 175
  precise, C-41–44
  preservation via hardware support, H-28–32
  RISC V, C-42–43, **C-42**
  stopping/restarting, C-41–42
  types and requirements, C-38–41, **C-40**
  unexpected sequences, C-70
Execute step
  Itanium 2, H-42
  TI 320C55 DSP, E-7
Execution, C-69, 198, 211
Execution address cycle (EX)
  data hazards requiring stalls, **C-18**
  data hazards stall minimization, C-14
  dynamic scheduling pipelines, C-66
  exception stopping/restarting, C-41
  floating point pipeline, C-46
  longer latency pipelines, C-51
  MIPS R4000 pipeline, C-58–59
  pipeline branch issues, C-35–36
  RISC exception, **C-42–43**, C-43
  RISC instruction set, C-5, **C-6**
  RISC pipeline, C-32–35
  simple RISC implementation, C-27
Execution time, 39
  Amdahl's law, 50
  application/OS misses, **B-59**
  cache performance, B-3
  central processing unit, B-3, B-5, **B-22**
  components, 400
  multiprocessor, 438
  multiprogrammed parallel "make" workload, **400**
  pipelining performance, C-3, C-8–10
  second-level cache size, B-32, **B-34**
  and stall time, B-21
  vector length, **G-7**
Expand-down field, **B-53**
Explicit parallelism, H-34–37
Explicit unit-stride, 333
Exponential back-off
  large-scale multiprocessor synchronization, I-17–18
  spin lock, **I-17**
Exponential distribution, D-27
Extended accumulator, A-3, K-30
Extended stack architecture, K-30

F
Fabrication cost, 67–68
Fabrication yield, 67–68
Failure. *See also* Mean time between failures (MTBF); Mean time to failure (MTTF)
  Berkeley's Tertiary Disk project, D-12
  definition, D-10
  dependability, 37–38
  dirty bits, D-61–64
  RAID
    reconstruction, D-55–57
    row-diagonal parity, **D-9**
  rates of disk subsystem, 51–52
  storage system, D-6–10
    components, D-43
  Tertiary Disk, D-13
Failures in time (FIT), 37
False sharing, 393–394, **398**
Fast Fourier transformation (FFT)
  characteristics, I-7
  distributed-memory multiprocessor, **I-32**
  example calculations, I-27–29
  symmetric shared-memory multiprocessors, I-22, **I-23**, I-25–26
Fat trees, F-34, **F-38**
Fault, 111
  definition, D-10
  dependability benchmarks, D-21
  programming mistakes, D-11
  Tandem Computers, D-13
Fault detection, 64
Fault-induced deadlock, F-45–46

Fault tolerance, F-70–72, F-98
  dependability benchmarks, D-21
  RAID, **D-7**
Fault-tolerant routing, F-70–71, F-98–99
FC. *See* Fibre Channel (FC)
FC-AL. *See* Fibre Channel Arbitrated Loop (FC-AL)
Feature Extraction, 573, **574**
Feature functional unit (FFU), 576
Feature maps, two-dimensional, 550
Feature size, 21
FEC. *See* Forward error correction (FEC)
Federal Communications Commission (FCC), D-15
FENCE in RISC V, 420–422
Fetch-and-increment, 413–414
  large-scale multiprocessor synchronization, I-20–21
  sense-reversing barrier, **I-21**
Fetch stage, TI 320C55 DSP, E-7
FFT. *See* Fast Fourier transformation (FFT)
Fibre Channel (FC), F-106
  file system benchmarking, **D-20**
  NetApp FAS6000 filer, D-41–42
Fibre Channel Arbitrated Loop (FC-AL), M-88, F-106
  block servers *vs.* filers, D-35
Fibre Channel Switched (FC-SW), F-106
Filers
  vs. block servers, D-34–35
  NetApp FAS6000 filer, D-41–43
  servers, SPEC benchmarking, D-20–21
Filters, radio receiver, **E-23**
Fine-grained multithreading, 243–244
Fingerprint, storage system, D-48
Finite-state machine, F-49, F-56–58
Firmware, network interfaces, F-7–8
First-in first-out (FIFO), B-9, **B-10**, 197
  definition, D-26
First-level caches
  cache optimization, B-30–35
  hit time/power reduction, 95–98
  interconnection network, F-74
  Itanium 2, **H-41**
  memory hierarchy, B-48–49, **B-48**
  parameter ranges, **B-42**
First-reference misses, B-23
Fixed-field decoding, C-5
Fixed length, 14
Fixed-point arithmetic, DSP, E-5–6
Flash memory
  disk storage, D-3–4
  embedded benchmarks, E-13
  memory hierarchy design, 92–93
  technology trends, 19
FLASH multiprocessor, M-62
Flexible chaining, 290–291
  vector processor, G-11
Flex point, 579
Floating-point (FP) operations, K-38–40
  addition
    denormals, J-26–27
    overview, J-21–25
    rules, **J-24**
    speedup, J-25–26
  chip comparison, **J-58**
  CPI, **C-64**
  data dependences, 171
  denormals, J-14–15, J-20–21, J-26–27
  double-precision, **C-63**
  DSP media extensions, E-10–11
  early computer arithmetic, J-64–65
  exceptions, J-34–35, C-41–42
  fused multiply-add, J-32–33
  IEEE 754, **J-16**
  integer conversions, J-62
  Itanium 2, **H-41**
  iterative division, J-27–31
  latency, **C-61**, C-63, **177**
  and memory bandwidth, J-62
  micro-op fusion, 254
  MIPS R4000 pipeline, C-60–61, **C-60**
  mispeculation, **239**
  multiplication
    denormals, J-20–21
    examples, **J-19**
    overview, J-17–20
    rounding, **J-18**, J-19
  multiplication precision, J-21
  multiply and add operation, **C-62**
  number representation, J-15–16
  overflow, **J-11**
  overview, J-13–14
  performance, 308
  pipeline scheduling, 178
  precisions, J-33–34
  programs, 101–102
  register file, **C-50**
  remainder, J-31–32
  result stalls, C-61, **C-64**
  RISC exception, C-43
  RISC multicycle operations, C-45–55
  RISC pipeline, C-45–55, **C-47–48**, **C-57**
  RISC-V, A-40–41
  special values, J-14–15, **J-16**
  square root, 51
  static branch prediction, **C-23**
  structural stalls, C-61, **C-64**
  Tomasulo's algorithm, **198**
  underflow, J-36–37, J-62
  vector chaining, G-11
Floating-point registers (FPRs)
  IA-64, H-34
  IBM Blue Gene/L, I-42
Floating Point Systems AP-120B, M-30
Floppy disks, M-85–86
Flow-balanced state, **D-24**
Flow control
  and arbitration, F-22
  interconnection networks, F-9–12
Fluent, F-80–81
Flush, branch penalty reduction, C-19
Forget gate, 553
Form factor, interconnection networks, F-9–10
FORTRAN
  compiler vectorization, G-14, **G-15**
  dependence analysis, H-6
  integer division/remainder, **J-12**
  performance measurement history, M-6
Forward error correction (FEC), E-5–7
Forwarding, C-14
  arithmetic-logical units, **C-36–37**
  data hazards stall minimization, C-14–15, **C-15–16**
  load instruction, **C-17**
  longer latency pipelines, C-49–52
  table, F-56–58
Forward path, cell phones, E-24
Fourier-Motzkin algorithm, M-32
Fourier transform, DSP, E-5

Four-way set associativity, B-23
FPGA, 568–569
  Catapult, 567
  Feature Extraction, **574**
FP operations. *See* Floating-point (FP) operations
Fragmentation problem, 114
Frame pointer, K-57
Free-form expressions, 573–574
Freeze, branch penalty reduction, C-19
Frequency modulation (FM), wireless neworks, E-21
Front-end stage, Itanium 2, H-42
FU. *See* Functional unit (FU)
Fujitsu Primergy BX3000 blade server, F-89
Fujitsu SPARC64 X+, 389, 426, 429
  feature, **427**
  performance, 429–431, **432**
Fujitsu VP100, M-48
Fujitsu VP200, M-48
Full access
  dimension-order routing, F-47–48
  interconnection network topology, F-30
Full adders, J-2–3, **J-3**
Full-duplex mode, F-23
Fully associative cache, B-7–9, B-12, 81
Fully connected layer, 549
Fully connected topology, F-35–36, **F-35–36**
Functional hazards, 266–273
Functional unit (FU), C-46
  execution slots, superscalar processors, 244–245, **244**
  Itanium 2, H-41–43
  latency, **C-47**, 177
  OCNs, F-3
Function pointers, A-18
Fused multiply-add, floating point, J-32–33
Future file approach, C-54

G
Gates, 553
Gateways, Ethernet, F-83
Gather-scatter, A-31–32, 301–302, 352
  sparse matrices, G-13–14
GE 645, M-9–10
GeForce 8800, M-52
General-Purpose Computing on GPUs (GPGPU), M-52
General-purpose electronic computers, M-2–4
General-purpose registers (GPRs)
  architectures, A-3
  IA-64, H-38
Geometric mean, 46
Gibson mix, M-6
Global address space, B-52
Global code scheduling
  example, **H-16**
  parallelism, H-15–23
  superblock scheduling, H-21–23, **H-22**
  trace scheduling, H-19–21, **H-20**
Global common subexpression elimination, A-26
Global data area, A-29
Global Environment for Network Innovation (GENI), F-102
Global miss rate, B-31
Global optimizations, A-26
Global Positioning System, CDMA, E-25
Global predictors, 184–188
Global scheduling algorithms, 219–220
Global system for mobile communication (GSM), cell phones, E-25
Goldschmidt's division algorithm, J-29–30
Goldstine, Herman, M-2–3
Google
  clusters history, M-63
  containers, M-76
Google App Engine, M-75–76
Google Clusters, 94
  power consumption, F-89
Google File System (GFS), 474
Google Translate, 4, 7, 40–45
Google WSCs
  airflow, 506
  availability zones, **498**
  cooling, 506–508
  generators, **505**
  networking, 510–511
  network switches, **502**
  network traffic, **501**
  on-site substation, **504**
  power distribution, 504–506
  power utilization efficiency of, **485**
  racks, 509–510, **509**, **512**
  servers, **505**, 512–513, **513**
  switch gear, **505**
  transformers, **505**
Gordon Bell Prize, M-58
GPGPU. *See* General-Purpose Computing on GPUs (GPGPU)
GPRs. *See* General-purpose registers (GPRs)
Gradual underflow, J-15, J-36
Grain size, 370
Grant phase, arbitration, F-49–50
Graph coloring, A-27
Graphical Processor Units (GPUs)
  computing history, M-52–53
  historical background, M-50–51
  scalable, M-51
Graphics data RAMs (GDRAMs), 90
Graphics-intensive benchmarks, 41
Graphics memory, 346
Graphics pipelines, M-51–52
Graphics processing unit (GPU), 10
  conditional branching, 323–326
  DNN and CPUs *vs.*, 595–602
  embedded *vs.* server, 346–353
  fallacy, 353
  multimedia SIMD and MIMD *vs.*, 347–353
  multimedia SIMD computers *vs.*, 335
  NVIDIA computational structures, 313–320
  NVIDIA GPU instruction set architecture, 320–323
  NVIDIA GPU memory structures, 326–328, **327**
  Pascal GPU architecture, 328–331
  programming, 310–313
  quick guide, **314**
  vector architectures *vs.*, 331–334, **332**
  vector kernel implementation, 357–359
Graphics synchronous DRAMs (GSDRAMs), 90
Graphics Synthesizer, Sony PlayStation 2, E-16–18, **E-16**

Greatest common divisor (GCD), 342–343
  test, loop-level parallelism dependences, H-7
Grid computing, M-75
Grid mapping, 315, **316**
Grid topology, F-36–38
Gshare predictors
  tagged hybrid *vs.*, **190**
  2-bit predictor, 184, **186**, 262
Guest domains, 126
Guest virtual machine, 121

H

Half adders, J-2–3
Half-duplex mode, F-23
Half-precision floating-point arithmetic, 329
Halo, 584–585
HAMR, 19
Handshaking, interconnection networks, F-10
Hard cores, Cortex-A53, 130
Hard errors, memory hierarchy design, 93
Hard real-time systems, definition, E-3–4
Hardware, 17
  compiler scheduling support, M-32–33
  compiler speculation support
    memory references, H-32
    overview, H-27
    preserving exception behavior, H-28–32
  designing, 17–18
  for exposing parallelism, H-23–27
  faults, D-11
  interconnection networks, F-8
  pipeline hazard detection, **C-34**
Hardware-based speculation, 208–217
  data flow execution, 209
  definition, 208
  disadvantage, 241
  instruction execution step, 211–212
  key ideas, 208
  reorder buffer, 209–212, 214–215
  vs. software speculation, 240–241
  write result, 217
Hardware prefetching, 109–111, 148–164
Hardware primitives, 412–414
  large-scale multiprocessor synchronization, I-18–21
Harvard architecture, M-3–4
Hazards. *See also* Data hazards
  control hazards, C-11
  data (*see* Data hazards)
  definition, C-10–11
  detection, hardware, **C-34**
  functional, 266–273
  instruction set complications, C-45
  longer latency pipelines, C-49–52
  read after write, C-12–14
  structural (*see* Structural hazards)
  write after read, C-12
  write after write, C-12
Header
  messages, F-7
  packet format, **F-7**
  switch microarchitecture pipelining, F-64
  TCP/IP, F-87–89
Head-of-line (HOL) blocking, F-59–61, **F-60**
Heap, A-29
HEP processor, M-35
Heterogeneity, DSAs, 592–594
Heterogeneous architecture, 282
Hewlett-Packard AlphaServer, F-104–105
Hewlett-Packard PA-RISC
  EPIC approach, M-33
  floating-point precisions, J-33
  MAX2, multimedia support, **E-11**
Hewlett-Packard RISC microprocessors, G-26
Hewlett Packard server, WSCs, 476
Hewlett-Packard's PA-RISC, K-3
Hidden layers, 546–547
High bandwidth memory (HBM), 346
  cache optimization, 114–117
  memory hierarchy design, 91
  Pascal GPU architecture, 329
Higher-radix division, J-54–55
Higher-radix multiplication, integer, J-48
High-level language computer architecture (HLLCA), M-20
High-level optimizations, A-26
Highly parallel memory systems, 150–153
High-order functions, A-18
High-performance computing (HPC), 466
  vector processor history, G-27–28
High Performance Fortran (HPF)—programs, 422
High-speed chip-to-chip interconnect, 329
Hillis, Danny, M-46, M-56, M-58–59, M-76
Histogram, D-26
History file approach, C-54
Hitachi S810, M-48
Hit time, B-15–16
  address translation, 83
  first-level caches, 95–98
  latency, **115**
  memory hierarchy design, 82
  reducing, 94
  reduction, B-36–40
  way prediction, 98–99
HLLCA. *See* High-level language computer architecture (HLLCA)
Home node, 406
Hop count, F-30
Hops, F-36, **F-40**
Host
  NVIDIA GPU memory structures, 327
  virtual machine, 121
Host channel adapters (HCAs), F-90
  historical background, M-89
Hot aisles, 506, **506**
Hot swapping, F-71–73
HPC. *See* High-performance computing (HPC)
HP Precision Architecture, integer arithmetic, J-11–12
HP ProLiant BL10e G2 blade server, F-89
HPSm, M-31
Hybrid predictors, 184
Hypercube networks, F-44, F-96
HyperTransport, F-67
  NetApp FAS6000 filer, D-42
Hypervisor. *See* Virtual machine monitor (VMM)

I

IAS machine, M-3, M-5
IBM
  BlueGene, 370
  Chipkill, 94
  cluster history, M-62, M-74
  computer history, M-5
  early VM work, M-10
  IBM 360, address space, B-58
  IBM 370 architecture, 124
  magnetic storage, M-85–86
  multiple-issue processor development, M-29–30
  RAID history, M-87
IBM 360
  architects, M-10
  computer architecture definition, M-18
  I/O bus history, M-89
  memory hierarchy development, M-9–10
  parallel processing debates, M-58
IBM 360/85, M-11, M-29
IBM 360/91
  early computer arithmetic, J-63
  history, M-29
  speculation concept origins, M-31
IBM 370
  early computer arithmetic, J-63–64
  integer overflow, **J-11**
  vector processor history, G-27
IBM 370/158, M-6–7
IBM 650, M-5
IBM 701, M-5
IBM 702, M-5
IBM 704, M-5, M-27–28
IBM 705, M-5
IBM 801, M-20
IBM 3081, M-61
IBM 7030, M-27–28
IBM 360/370 architecture, K-69–70
  branches and special loads and stores—RX format, K-72
  branches and status setting R-R instructions, K-71
  branches/logical and floating-point instructions—RX format, K-71
  definition, K-69
  360 detailed measurements, K-70–74
  historical perspective and references, K-75
  integer/logical and floating-point R-R instructions, K-70
  measurements, K-70–74
  RS and SI format instructions, K-72
  SS format instructions, K-73
IBM AS/400, M-87
IBM Blue Gene/L, F-4, **I-44**
  cluster history, M-64
  computing node, I-42–44, **I-43**
  as custom cluster, I-41–44, **I-43–44**
  deterministic *vs.* adaptive routing, **F-53–56**
  parallel processing debates, M-59
  system area network, F-76–77
  3D torus network, F-39
IBM 3840 cartridge, M-85
IBM 9840 cartridge, M-85
IBM CoreConnect
  cross-company interoperability, F-68
  OCNs, F-3
IBM eServer p5 multiprocessor
  benchmarks, 440
  cost-performance, 440, **441**
IBM Federation network interfaces, F-18
IBM Power 1, M-31
IBM Power 2, M-31
IBM Power 4
  multithreading history, M-36
  recent advances, M-35
IBM Power 5, 424
  Itanium 2 comparison, **H-43**
  multithreading history, M-36
IBM Power 8, 371, 389–390, 426
  design, 429
  feature, **427**
  on-chip organizations, **428**
  performance, 431–432, **432**
IBM Power processors
  branch-prediction buffer, **C-25**
  characteristics, **265**
IBM Pulsar processor, M-35–36
IBM RP3, M-61
IBM RS/6000, M-58
IBM RT-PC, M-21
IBM SAGE, M-89
IBM Stretch, M-6
IBM 3090 Vector Facility, G-27
IBM zSeries, G-27
IC. *See* Instruction count (IC)
I-cache, way prediction, 98–99
ID. *See* Instruction decode (ID)
Ideal pipeline CPI, 169
IDE disks, Berkeley's Tertiary Disk project, D-12
Idle Control Register (ICR), TI TMS320C55 DSP, E-8
Idle domains, TI TMS320C55 DSP, E-8
IEEE arithmetic
  floating point, J-13–14
    addition, J-21–27
    exceptions, J-34–35
    multiplication, J-17–21
    remainder, J-31–32
    underflow, J-36–37
  historical background, J-63–65
  iterative division, J-30
  NaN, J-14
  rounding modes, **J-20**
  single-precision numbers, J-15
  –*x vs.* 0 –*x*, J-62
IEEE 754 floating-point standard, **J-16**
IEEE 1394, Sony PlayStation 2 Emotion Engine case study, E-15
IEEE standard 802.3 (Ethernet), F-82
  LAN history, F-82
IF cycle. *See* Instruction fetch (IF) cycle
Illiac IV, M-45, M-55, F-104
ILP. *See* Instruction-level parallelism (ILP)
Image processing units (IPUs), 580–582
Image signal processors (ISPs), 580–582
  hardwired predecessors of, 580–581
  interconnection of, **582**
Immediate addressing, K-52
IMPACT, M-33
Implicit unit stride, 333
Imprecise exceptions, 194
Inclusion, 383
  drawback, B-35
  implementation, 423–424
  L1 caches, 423–424

- Inclusion *(Continued)*
  - L2 caches, 423–424
  - L3 caches, 424
  - memory hierarchy history, M-12
  - multilevel, B-34, 423
  - property, 78
- Indexed addressing, K-34, K-53
- Indexes
  - address translation during, B-36–40
  - Opteron data cache, B-13
  - recurrences, H-12
- Index field, B-8–9
- Index vector, 301–302
- Indirect addressing, K-52
- Indirect jumps, branch prediction, 232–234
- Indirect networks, F-32–33
- Inexact exception
  - floating-point arithmetic, J-35
  - floating-point underflow, J-36
- InfiniBand, F-62, F-68, F-77–81, **F-79**
  - cluster history, M-64
- Infinite population model, D-30
- Initiation rate, 290
- Inktomi, M-63, M-74–75
- In-order commit, speculation concept origins, M-31
- In-order execution
  - average memory access time, B-18
  - cache miss, B-2
  - IBM Power processors, **265**
- Input buffered switch, F-56
- Input gate, 553
- Input-output buffered switch, F-56, **F-58**, **F-65**
- Instruction cache
  - AMD Opteron example, B-15, **B-15**
  - application/OS misses, **B-59**
  - branch prediction, C-24
  - TI TMS320C55 DSP, E-8
- Instruction commit, C-43–44, 209, 235
- Instruction count (IC), B-4, 53
  - cache performance, B-15–16
  - processor performance equation, 52
  - RISC history, M-23
- Instruction decode (ID)
  - branch hazards, C-18
  - data hazards requiring stalls, **C-18**
  - dynamic scheduling, C-66, 193
  - longer latency pipelines, C-50–51
  - MIPS R4000 pipeline, C-56
  - pipeline branch issues, C-35–36
  - RISC classic pipeline, C-8
  - RISC instruction set, C-5, **C-6**
  - RISC pipeline, C-32–34, **C-35**
  - simple RISC implementation, C-27
- Instruction delivery, 228–240
  - stage, Itanium 2, H-42
- Instruction fetch (IF), 253
  - bandwidth, 228–232, **229–230**
  - cycle
    - ARM Cortex-A53, 249–250
    - branch hazards, C-18
    - branch-prediction buffer, C-24
    - data hazards requiring stalls, **C-18**
    - exception stopping/restarting, C-41
    - MIPS R4000 pipeline, C-56
    - RISC exception, **C-42–43**, C-43
    - RISC instruction set, C-4, **C-6**
    - RISC pipeline, C-31–33
    - simple RISC implementation, C-27
  - units, integrated, 233–234
- Instruction formats
  - addressing modes and, K-6–9
  - high-level language computer architecture, M-20
  - IA-64 ISA, H-34–38, **H-39**
- Instruction groups, IA-64, H-34
- Instruction issue, C-33–34
  - Itanium 2, H-41–43
- Instruction-level parallelism (ILP), 5, 10, 368, 370
  - aggressive compiler-based approaches, 168
  - approaches, 168
  - branch prediction
    - correlating branch predictors, 182–184
    - Intel Core i7, 190–191
    - specialized, 232–234
    - tagged hybrid predictors, 188–190, **188**, **190**
    - tournament predictors, 184–188, **186**
  - branch-prediction buffer, C-25, **C-25**
  - clock cycles per instruction, 168–169
  - compiler scheduling, M-32
  - compiler techniques, 176–182
  - concepts, 169–170
  - control dependences, 174–176
  - data dependences, 170–172
  - data flow limit, M-34
  - data hazards, 173–174
  - definition, 168
  - dynamic scheduling, 222–227
    - advantages, 191–192
    - data hazards, 191–201
    - out-of-order execution, 193–194
    - Tomasulo's algorithm, 195–208, **205–206**
  - early studies, M-33–34
  - exploitation methods, H-21–23
  - exploitation of, 2
  - exploitation statically, H-2
  - exposing with hardware support, H-23–27
  - IA-64, H-32
  - loop unrolling, 177–182
  - microarchitectural techniques case study, 266–273
  - multiple-issue processors, M-31, 218–227
    - advantages, 221–222
    - challenges, 182, 221–222
    - characteristics, **219**
    - dynamically scheduled processor, 222, 224
    - EPIC approach, 221
    - microarchitectural techniques case study, 266–273
    - with speculation, **223**
    - superscalar, 218, 223
    - VLIW approach, 218–222, **220**
  - multithreading history, M-36
  - name dependences, 172–173
  - pipeline scheduling, 177–182
  - scaling, 442
  - speculation, 222–227
    - address aliasing prediction, 239–240
    - advanced techniques, 228–240
    - advantages, 237–238
    - challenge of issues per clock, 236–237

        control dependence, 175–176
        disadvantages, 238
        and energy efficiency, 238–239
        exception handling, 199
        execution, 241
        hardware *vs.* software, 240–241
        microarchitectural techniques case study, 266–273
        multiple branches, 238
        register renaming *vs.* ROB, 234–236
    static scheduling, 218–222
    TI 320C6x DSP, E-8
Instruction path length, 52
Instruction prefetch, 234
Instruction register (IR)
    RISC pipeline, C-31–32
    simple RISC implementation, C-27
Instruction set architecture (ISA), 12–17. *See also* Intel 80x86 processors; Reduced Instruction Set Computer (RISC)
    byte-addressed computers, **A-8**
    changes, A-46–47
    classes, **A-4**
    classifying, A-3–6
    class of, 12
    complications, C-43–45
    computer architecture definition, M-18–19
    Cray X1, G-21–22
    cross-cutting issues, 126–127
    encoding, 14, A-21–24, **A-22**
    fallacies and pitfalls, A-42–55
    first vector computers, M-49
    general-purpose register computers, **A-6**
    high-level language computer architecture, M-20
    IA-64
        instruction formats, H-34–37, **H-39**
        instructions, **H-35–37**
        instruction set basics, H-38
        overview, H-32–40
        predication and speculation, H-38–40
        register model, H-33–34
    memory addressing, A-7–13
    memory and total operands, **A-5**
    MIPS
        RISC history, M-20–23, **M-22**
        stack architectures, M-17–18
    operands, **A-4**, A-13–15
    operations in, A-15–16
    optimizations impact on performance, A-27
    performance and energy efficiency, 258
    Pixel Visual Core, 587–588
    register allocation, A-27
    RISC-V, A-34
    TPU, 559
    virtual machine, 120–121
    for virtual machine, 122–123
Instruction set, extension, 124–125
Instructions per clock (IPC), 52, 169
Integer arithmetic
    addition speedup
        carry-lookahead, J-37–41
        carry-lookahead circuit, **J-38**
        carry-lookahead tree, **J-40**
        carry-lookahead tree adder, **J-41**
        carry-select adder, J-43–44, **J-43–44**
        carry-skip adder, J-41–43, **J-42**
        overview, J-37
    division
        radix-2 division, **J-55**
        radix-4 division, **J-56**
        radix-4 SRT division, **J-57**
        with single adder, J-54–57
        SRT division, J-45–47, **J-46**, J-55–57
    FP conversions, J-62
    language comparison, **J-12**
    multiplication
        array multiplier, **J-50**
        Booth recoding, **J-49**
        even/odd array, **J-52**
        many adders, J-50–54, **J-50**
        multipass array multiplier, **J-51**
        signed-digit addition table, **J-54**
        with single adder, J-47–49, **J-48–49**
        Wallace tree, **J-53**
    multiplication/division, shifting over zeros, J-45
    overflow, **J-11**
    Radix-2 multiplication/division, J-4–7, **J-4**
    restoring/nonrestoring division, J-5, **J-6**
    ripply-carry addition, J-2–3
    signed numbers, J-7–10
    SRT division, J-45–47, **J-46**
    systems issues, J-10–13
Integer operations
    ARM Cortex-A53, **249**
    data dependences, 171
    Itanium 2, **H-41**
    mispeculation, **239**
    RISC pipeline, C-45–55
    stalls, C-55
    static branch prediction, C-22, **C-23**
Integer registers, IA-64, H-33–34
Integrated branch prediction, 233
Integrated circuits
    basics, cell phones, E-24, **E-24**
    cost of, 31–35
    dependability, 36–38
    logic technology, 19
    power and energy in, 23–29
Intel 80286, M-9–10
Intel Core i7, 100
    branch prediction, 190–191
    buffers and queues, **256**
    hardware prefetching, 110
Intel Core i7 920
    characteristics, **259**
    clock cycles per instruction, 256, **257**
    misprediction rate, **192**
    relative performance and energy efficiency, **260**
Intel Core i7 6700, 55–56
    clock cycles per instruction, 256, **257**
    memory hierarchy design, 133–142, **134–135**
    misprediction rate, **192**
    multiple-issue processors, 247, 252–257
    performance, 138–142, 255–257
    pipeline structure, **254**
Intel Core i7 microprocessor
    die, **32**
    fallacy, 61
    floorplan, **32**

Intel Core i7 920 multicore computer, **309**
Intel Crest, 579, **580**
Intel 8087, floating point remainder, J-31
Intel Haswell CPU Roofline, **599**
Intel i7, 388, **395**
Intel i7 920
  performance and energy efficiency, 434–437
  simultaneous multithreading, 246
Intel i860, M-30, M-32, M-50, M-60–61
Intel IA-32 architecture
  call gate, **B-53**, B-54
  descriptor table, B-52
  instruction set complications, C-45
  OCNs, F-3
  segment descriptor, B-52, **B-53**
  segmented virtual memory, B-51–54
Intel IA-64 architecture
  compiler scheduling history, M-32–33
  conditional instructions, H-27
  explicit parallelism, H-34–37
  historical background, M-33
  ISA
    instruction formats, H-34–37, **H-39**
    instructions, **H-35–37**
    instruction set basics, H-38
    overview, H-32–40
    predication and speculation, H-38–40
  Itanium 2 processor
    instruction latency, **H-41**
    overview, H-40–41
    performance, H-43, **H-43**
  parallelism exploitation statically, H-2
  register model, H-33–34
  RISC history, M-23
  software pipelining, H-14–15
  synchronization history, M-64–65
Intel iPSC 860, M-60–61
Intel Itanium, 168
  instruction-level parallelism, 261–262
  sparse matrices, G-13
  speculation, 241
Intel Itanium 2
  IA-64
    functional units and instruction issue, H-41–43
    instruction latency, **H-41**
    overview, H-40–41
    performance, H-43, **H-43**
Intellectual property, DSAs, 593
Intelligent devices, historical background, M-88
Intel Paragon, M-60–61, F-96
Intel Pentium 4, **110**, 261–262
  Extreme, M-35
  Itanium 2 comparison, **H-43**
  multithreading history, M-36
Intel Pentium II, M-35
Intel Pentium III, power consumption, F-89
Intel Pentium M, F-89
Intel Pentium MMX, multimedia support, **E-11**
Intel Pentium Pro, M-35
Intel Pentium processors
  early computer arithmetic, J-65
  vs. Opteron memory protection, B-57
  segmented virtual memory, B-51–54
Intel processor, **261**
  instruction set extensions, 125
  multiple processors, 5
  power consumption, F-89
Intel Teraflops processors, OCNs, F-3
Intel Thunder Tiger 4 QsNet^{II}, F-67
Intel 80x86
  comparative operation measurements, K-45–48
  floating-point operations, K-38–40
  instruction encoding, K-40–43
  integer operations, K-35–37
  measurements of instruction set usage, K-44–48
  operand addressing, K-44–45
  processors
    address space, B-58
    integer overflow, **J-11**
    memory hierarchy development, M-9–10
    protection structure, B-50
  registers and data addressing modes, K-32–35
  SPECint92 programs, **K-44**, **K-46**, **K-48–50**
Intel x86, conditional instructions, H-27
Intel Xeon, F-80, **354**, 387
Intel Xeon E7, 426
Interactive workloads, 467
Interarrival times, D-30
Interconnection networks
  adaptive routing, F-97
  adaptive routing and fault tolerance, F-98
  arbitration, F-49–51
  basic characteristics, F-2, **F-21**
  bisection bandwidth, F-93
  commercial
    congestion management, F-68–70
    connectivity, F-67
    cross-company interoperability, F-67–68
    DECstation 5000 reboots, **F-73**
    fault tolerance, F-70–72
  communication bandwidth, I-3
  compute-optimized processors *vs.* receiver overhead, F-92
  definition, F-2
  density- *vs.* SPEC-optimized processors, F-89
  device example, **F-3**
  direct *vs.* high-dimensional, F-96
  domains, F-3–4, **F-3**
  Ethernet, F-82–84
  examples, F-73–85
  HOL blocking, F-59–61
  IBM Blue Gene/L, I-43
  InfiniBand, F-77–81
  LAN, F-82–84
  link bandwidth, F-94
  memory hierarchy interface, F-91–92
  mesh network routing, **F-47**
  MIN *vs.* direct network costs, F-96
  multi-device connections
    basic considerations, F-20–21
    effective bandwidth *vs.* nodes, **F-29**
    latency *vs.* nodes, **F-28**
    performance characterization, F-25–30

    - shared-media networks, F-23–24
    - shared- *vs.* switched-media networks, **F-23**, F-25
    - switched-media networks, F-24–25
    - topology, routing, arbitration, switching, F-21–22
  - OCN, **F-27–29**
  - protection, F-91
  - routing, F-22, F-44–56
  - routing/arbitration/switching impact, F-21–22
  - SAN characteristics, **F-27–29**
  - software overhead, F-96
  - storage area networks, F-106–108
  - switching, F-51–52
  - switch microarchitecture, F-56–66
  - switch *vs.* NIC, **F-90**
  - system area networks, F-104–106
  - system/storage area network, F-77–81
  - top-level architecture, **F-75**
  - topology, F-30–44
  - two-device interconnections
    - basic considerations, F-6
    - effective bandwidth *vs.* packet size, **F-19**
    - example, **F-6**
    - interface functions, F-6–9
    - performance, F-13–20
    - structure and functions, F-9–12
    - virtual channels and throughput, F-47–48
    - WAN, F-84–85
    - wormhole switching performance, F-52
    - zero-copy protocols, F-95
- Intermittent faults, D-11
- Internal fragmentation, B-47
- International Computer Architecture Symposium (ISCA), M-12
- International Mobile Telephony 2000 (IMT-2000), cell phone standards, E-25
- International Technology Roadmap for Semiconductors (ITRS), 58–59, **59**
- Internet Archive Cluster, D-36–41
  - containers, M-76
- Internet of Things (IoT), 6–7
- Internet Protocol (IP), F-85–89
  - cores, OCNs, F-3
  - routers, VOQs, F-31, F-103
- Internetworking, F-2, **F-84**, F-85–89
- Interprocedural analysis, H-10
- Interprocessor communication, large-scale multiprocessors, I-3–6
- Interrupt Enable (IE) flag, 127
- Invalidate protocol, 380
  - example, 385, **385**
  - implementation, 382–383
  - snooping coherence, **381**
- Invalid exception, floating-point arithmetic, J-35
- Inverted page table, B-44–45
- I/O bandwidth, D-15–16
- I/O benchmarks, response time restrictions, D-18
- I/O-bound, 121, 123–124
- I/O bus
  - historical background, M-88–89
  - point-to-point replacement, **D-34**
  - Sony PlayStation 2 Emotion Engine case study, E-15
- I/O cache coherency, 128–129
- I/O devices
  - address translation, B-38
  - historical background, M-88–89
  - performance, D-15–23
  - SANs, F-4
  - shared-media networks, F-23
  - switched networks, F-2
  - switch *vs.* NIC, F-90
  - write strategy, B-11
- I/O interfaces, storage area network history, F-107
- I/O network, F-67
- I/O processor (IOP)
  - first dynamic scheduling, M-28–29
  - Sony PlayStation 2 Emotion Engine case study, E-15
- I/O subsystems
  - design, D-59–61
  - interconnection network speed, F-92
  - *vs.* NIC, F-95
  - zero-copy protocols, F-95
- I/O systems
  - asynchronous, D-35–36
  - as black box, **D-24**
  - dirty bits, D-61–64
  - multithreading history, M-35
  - queing theory, D-23
  - queue calculations, D-29
  - random variable distribution, D-26
- IP block, DSAs, 593
- IPC. *See* Instructions per clock (IPC)
- IPoIB, F-81
- IR. *See* Instruction register (IR)
- ISA. *See* Instruction set architecture (ISA)
- iSCSI
  - NetApp FAS6000 filer, D-41–42
  - storage area network, F-106–107
- ISPs. *See* Image signal processors (ISPs)
- Issue logic, 236
- Issue stage
  - ID pipe stage, 194
  - instruction step, 197, 211
- Iterative division, floating point, J-27–31

J

- Java benchmark, 435–437, **435**, **437**
- Java language, dependence analysis, H-10
- Java Virtual Machine (JVM), early stack architectures, M-18
- Johnson, Reynold B., M-85
- Jumps, VAX, K-57
- Just-in-time (JIT), M-18

K

- Kahle, Brewster, M-76
- Kahn, Robert, F-102
- *k*-ary *n*-cubes, F-38–39
- Kendall Square Research KSR-1, M-61–62
- Kernels
  - EEMBC benchmarks, **E-12**
  - FFT, I-7
  - FORTRAN, compiler vectorization, **G-15**
  - LU, I-8
  - process, 40
    - DAG of, 582
    - Driver, 563
    - segmented virtual memory, B-51

- Kernels *(Continued)*
    - throughput computing, 350
    - vector kernel implementation, 357–359
    - virtual memory, 119

L

- LabVIEW, embedded benchmarks, E-13
- Lampson, Butler, F-103
- Large-scale multiprocessors
    - cache coherence implementation, I-34–35
        - deadlock and buffering, I-38–40
        - directory controller, I-40–41
        - DSM multiprocessor, I-36–37
    - characteristics, **I-45**
    - cluster history, M-63–64
    - example calculations, I-12–13
    - historical background, M-60–62
    - IBM Blue Gene/L, I-41–44, **I-43–44**
    - interprocessor communication, I-3–6
    - for parallel programming, I-2
    - scientific applications, I-6–12
        - distributed-memory multiprocessors, I-26–32, **I-28–32**
        - parallel processing, I-33–34
        - symmetric shared-memory multiprocessor, I-21–26, **I-23–26**
    - space and relation of classes, **I-46**
    - synchronization
        - mechanisms, I-17–21
        - performance, I-12–16
- Latency, 20. *See also* Response time
    - ALU, C-46–48
    - bandwidth, F-25–30
    - barrier synchronization, I-16
    - and cache miss, B-2
    - cluster history, M-74
    - communication mechanism, I-3
    - definition, D-15–16, C-46–48
    - deterministic *vs.* adaptive routing, **F-53–56**
    - distributed-memory multiprocessors, I-30, **I-32**
    - Flash memory, D-3
    - FP operations, **C-61**, C-63, **177**
    - functional units, **C-47**
    - hazards and forwarding, C-49–52
    - interconnection networks, F-13–20
    - Itanium 2 instructions, **H-41**
    - microarchitectural techniques case study, 266–273
    - OCNs *vs.* SANs, **F-28**
    - out-of-order execution, B-20–21
    - packets, F-13, **F-13**
    - performance trends, 20, **21**
    - snooping cache coherence, 447, **448**
    - Sony PlayStation 2 Emotion Engine, E-17
    - throughput *vs.* response time, D-16
    - utility computing, M-75
    - vector memory systems, G-9
    - vector start-up, **G-8**
- Latency-hiding techniques, 418
- L1 cache. *See also* First-level caches
    - address translation, B-46
    - Alpha 21164, **395**
    - ARM Cortex-A53, 251–252
    - data cache size, **402**
    - first-level caches, 95–98
    - inclusion, 423–424
    - Intel i7, **395**
    - memory hierarchy, **B-39**, B-48–49, **B-48**
    - miss rate, **402**
    - Opteron memory, **B-57**
- L2 cache. *See also* Second-level caches
    - Alpha 21164, **395**
    - ARM Cortex-A53, 251–252
    - cache optimization, **B-34**, B-35
    - IBM Blue Gene/L, I-42
    - inclusion, 423–424
    - Intel i7, **395**
    - memory hierarchy, **B-39**, B-48–49, **B-48**, **B-57**
    - memory system, 241
- L3 cache. *See also* Third-level caches
    - Alpha 21164, **395**
    - IBM Blue Gene/L, I-42
    - IBM Power8, 371
    - IBM Power processors, **265**
    - inclusion, 424
    - Intel i7, **395**
    - memory access cycle shift, 396–397, **397**
    - miss rate, 397–399, **398**
    - snoop bandwidth, 390
- Learning curve, 30
- Least common ancestor (LCA), F-48–49
- Least recently used (LRU)
    - AMD Opteron data cache, B-14
    - block replacement, B-9–10, **B-10**
    - memory hierarchy history, M-11
    - virtual memory block replacement, B-45
- Limit field, B-52
- Linear speedup, multiprocessor, 438–439, **440**
- Line Buffer Pool (LBP), 590
- Line locking, embedded systems, E-4
- Line, memory hierarchy, 81
- Link injection bandwidth
    - calculation, F-17
    - interconnection networks, F-26–27
- Link pipelining, F-16–17
- Link reception bandwidth, calculation, F-17
- Link register, K-25
- Linpack benchmark
    - cluster history, M-64
    - parallel processing debates, M-59
    - VMIPS performance, G-17–19
- Linux operating system, RAID benchmarks, **D-22**
- Liquid crystal display (LCD), Sanyo VPC-SX500 digital camera, E-19
- LISP, K-21–22
    - RISC history, M-20–21
- Little Endian
    - byte, A-7
    - interconnection networks, F-12
- Little's Law, 328
    - definition, D-24
    - server utilization calculation, D-29
- Livelock, network routing, F-45–46
- Liveness, control dependences, 176
- Livermore Fortran Kernels, M-6
- Load instruction
    - control dependence, 175
    - data hazards requiring stalls, C-17, **C-17**
    - RISC instruction set, C-5
- Load interlock, C-33–34, **C-35**
- Load memory data (LMD), C-28–29
- Load reserved, 413–414, 416
- Loads instruction, 199

Load stalls, C-61, **C-64**
Load-store architecture, A-3
Load-store instruction set architecture, C-5, 12, C-28
  RISC history, M-20
Local address space, B-52
Local area networks (LANs)
  characteristics, F-4
  cross-company interoperability, F-67–68
  effective bandwidth, F-18–19
  Ethernet as, F-82
  fault tolerance calculations, **F-72**
  InfiniBand, F-77–78
  interconnection network domain relationship, F-4, **F-5**
  latency and effective bandwidth, F-27–29
  offload engines, F-8
  packet latency, F-13–16, **F-13**
  shared-media networks, F-23
  time of flight, F-14
Local memory
  centralized shared-memory architectures, 377
  multiprocessor architecture, 371–373
  NVIDIA GPU memory structures, 326–327
Local miss rate, B-31
Local node, 406
Local optimizations, A-26
Local predictors, 184–188
Local scheduling techniques, 219–220
Local state, 377
Location counts, WSCs, 468
Locks
  caching, 415
  large-scale multiprocessor synchronization, I-18–21
  spin, 414–416
  using coherence, 414–417, **416**
Lockup-free cache, 100–104
Logical units, storage systems, D-34
Logical volumes, D-34
Long displacement addressing, K-52
Long Instruction Word (LIW)
  EPIC approach, M-33
  multiple-issue processor development, M-30, M-32
Long mode, K-32
Long short-term memory (LSTM) cells, 553, **554–555**
Loop branches prediction, 232–234
Loop-carried dependence, 289, 312, 337–339
  dependence distance, H-6
  example calculations, H-4–5
  loop-level parallelism, H-3
  recurrence form, H-5
Loop interchange, cache optimization, 107
Loop-level parallelism, H-2–12
  analysis, 337–339
  CUDA/NVIDIA term, **337–338**
  definition, 169–170
  dependent computations, 344–345
  detection and enhancement, H-2–12
    dependence analysis, H-6–10
    dependent computation elimination, H-10–12
  finding dependences, 339–344
  history, M-32–33
  SIMD, 170
Loop stream detection, 254
Loop unrolling
  limitation, 181
  and scheduling, 181–182
  software pipelining, H-12–15, **H-13**, **H-15**
Lossless networks
  definition, F-12
  switch buffer organizations, F-59–60
Lossy networks, definition, F-12
LRU. *See* Least recently used (LRU)
LU kernel
  characteristics, I-8
  distributed-memory multiprocessor, **I-32**
  symmetric shared-memory multiprocessors, I-22, **I-23**, I-25–26

M
MAC. *See* Multiply-accumulate (MAC)
Machine language programmer, M-18
Machine learning, 546, 573
Machine memory, virtual machine, 123
Macro-op fusion, 253
Magnetic disk technology, 19
Magnetic storage
  access time, D-3
  cost *vs.* access time, **D-3**
  historical background, M-85–86
Mail servers, D-20–21
Main memory, B-2–3, 377, 400
  block identification, B-44–45
  block placement, B-44
  cache function, B-2
  vector processor, G-25
  vs. virtual memory, B-41
  write strategy, B-45–46
MapReduce
  AWS, 495, **495**
  WSCs, 471–476, **472**
Mark-I, M-3–4, M-6
Mark-II, M-3–4
Mark-III, M-3–4
Mark-IV, M-3–4
MasPar, M-46, M-56
Massively parallel processors (MPPs)
  characteristics, **I-45**
  cluster history, M-62–63, M-74
  system area network, F-104
Matrix300 kernel, prediction buffer, **C-25**
Matrix multiplication, LU kernel, I-8
Matrix multiply unit, 557–558, **558**, **562**
Mauchly, John, M-2–3, M-5, M-20
Maximum transfer unit, network interfaces, F-8
McCreight, Ed, F-103
McFarling's gshare predictor, 184
MCP operating system, M-17–18
Mean time between failures (MTBF), 37
Mean time to failure (MTTF)
  computer system power consumption case study, 69–71
  dependability, 37–38
    benchmarks, D-21
  disk arrays, D-6
  fallacy, 62
  I/O subsystem design, D-59–61
  RAID, M-86–87
  RAID reconstruction, D-55–57
  TB-80 cluster, D-41
  WSCs, 468

Mean time to repair (MTTR), 37
  dependability benchmarks, D-21
  disk arrays, D-6
  RAID 6, D-8–9
  RAID reconstruction, D-55–56
Mean time until data loss (MTDL), D-55–57
Media extensions, DSPs, E-10–11
Media, interconnection networks, F-9–12
Mellanox MHEA28-XT, F-80
Memory access
  ALUs, data forwarding, **C-36–37**
  cache hit calculation, B-5
  Cray Research T3D, F-91, **F-91**
  cross-cutting issues, 127–128
  data hazards stall minimization, C-14, **C-16**
  distributed-memory multiprocessor, **I-32**
  exception stopping/restarting, C-41
  instruction set complications, C-43–44
  integrated instruction fetch units, 234
  longer latency pipelines, C-51
  MIPS R4000 pipeline, C-59
  multicycle FP operations, C-52
  RISC classic pipeline, C-8
  RISC exception, **C-42–43**, C-43
  RISC instruction set, C-5, **C-6**
  RISC pipeline, C-32–35, **C-36**
  simple RISC implementation, C-28
  vector architectures, **G-10**
Memory addressing, 13
  addressing modes, A-8–11, **A-10**
  compiler-based speculation, H-32
  displacement addressing mode, A-11–12
  immediate/literal, A-12, **A-12**
  interpreting, A-7–8
  vector architectures, **G-10**
Memory bandwidth, 350, 356
Memory banks. *See also* Banked memory
  example, 301
  vector architecture, 298–299
  vector systems, G-9–11
Memory bus (M-bus), 377
  interconnection networks, F-91–92
Memory consistency, 377–378, 417–422
  case study, 456–458
  compiler optimization, 422
  development of models, M-64–65
  programmer's view, 418–419
  relaxed consistency models, 419–422, **421**
  sequential consistency, 417
  speculation to hide latency, 422–423
Memory-constrained scaling, I-33–34
Memory hierarchy, F-91–92
  address space, B-57
  block identification, B-8–9
  block placement, B-7–8, **B-7**
  block replacement, B-9, **B-10**
  cache optimizations, B-22–40, **B-40**
    hit time reduction, B-36–40
    miss categories, B-23–25
    miss penalty reduction, B-30–36
    miss rate reduction, B-26–30, **B-27**
  cache performance, B-3–6, B-15–16, **B-40**
    average memory access time, B-17–20
    equations, **B-22**
    example calculation, B-16–17
    miss penalty, B-20–21
    out-of-order execution, B-20–21
  development, M-9–12
  levels in slow down, **B-3**
  Opteron data cache example, B-12–15, **B-13**, **B-15**
  Opteron L1/L2, **B-57**
  OS and page size, B-58
  questions, B-6–12
  terminology, B-2
  virtual address to L2 cache, **B-39**
  virtual memory, B-2–3, B-40–49
    address space, B-12, **B-41**, B-44, B-55
    address translation, B-46, **B-47**
    caches and, B-42–43, **B-42**, B-48–49, **B-48**
    classes, B-43
    Intel Pentium *vs.* AMD Opteron, B-57
    paged example, B-54–57
    page size selection, B-46–47
    parameter ranges, **B-42**
    Pentium *vs.* Opteron protection, B-57
    protection, B-49–50
    questions, B-44–46
    segmented example, B-51–54
  write strategy, B-45–46
  WSCs, 479–482, **479**
Memory hierarchy design
  ARM Cortex-A53, 129–131, **130**
  basics of, 81–84
  cache optimization
    advancement, **117**
    cache misses, 112–113
    compiler-controlled prefetching, 111–114
    compiler optimizations, 107–109
    critical word first, 104–105
    early restart, 104–105
    energy consumption, **97**
    floating-point programs, 101–102
    hardware prefetching, 109–111
    HBM packaging, 114–117
    hit time/power reduction, 95–98
    multibanked caches, 99–100
    nonblocking caches, 100–104
    pipelined access, 99–100
    way prediction, 98–99
    write buffer merging, 105–106, **106**
  Cortex-A53 performance, 132
  C program evaluation, **151**
  cross-cutting issues
    autonomous instruction fetch units, 127
    case study, 150–153
    coherency of cached data, 128–129
    protection, virtualization, and instruction set architecture, 126–127
    special instruction caches, 128
    speculation and memory access, 127–128
  Intel Core i7 6700, 133–142, **134**
  in personal mobile device, **79**
  pitfall, 143

  technology and optimizations
    dependability, 93–94
    DRAM technology, 85–87
    Flash memory, 92–93
    GDRAMs, 90
    phase-change memory technology, 93
    SRAM technology, 85
    stacked/embedded DRAMs, 91
    synchronous DRAM, 87–90
  virtual machine
    hardware management, 121
    impact on virtual memory, 123–124
    instruction set architecture for, 122–123
    protection via, 120–122
    software management, 121
  virtual machine monitor
    instruction set extension, 124–125
    requirements, 122
    Xen virtual machine, 126
  virtual memory
    instruction set extension, 124–125
    protection via, 119–120
    virtual machines impact on, 123–124
Memory latency, 85
Memoryless, D-27–28
Memory mapping, B-52
Memory-memory architecture, A-3
Memory protection
  exception, 175
  Pentium *vs.* Opteron, B-57
  processes, B-49–50
  safe calls, B-54
  segmented virtual memory, B-51–54
Memory stall cycles
  average memory access time, B-18
  definition, B-3–4, **B-22**
  miss rate calculation, B-6
  out-of-order execution, B-20
Memory system, 377–378
  coherency, 378
  Intel Core i7 6700 pipeline structure, **254**
  multiprocessor architecture, 369, 371–373
  page size changes, B-58
  speculative execution, 241
  vector architectures, G-9–11
  vector chaining, G-11
  virtual (*see* Virtual memory)
Memristor, 93
Mesh interface unit (MIU), F-74
Mesh network, F-43, **F-47**
Mesh topology, F-74
MESIF protocol, 388
MESI protocol, 388, 449
Message ID, packet header, F-8, F-17
Message-passing communication
  advantages, I-5–6
  historical background, M-60–61
Message Passing Interface (MPI)
  function, F-8
  InfiniBand, F-80–81
  lack in shared-memory multiprocessors, I-5
Message-passing protocols, 373
Messages
  adaptive routing, **F-64**
  interconnection networks, F-6–9
  zero-copy protocols, F-95
MFLOPS. *See* Millions of floating-point operations per second (MFLOPS)
Microarchitecture, 17, **272**
  case study, 266–273
  Cray X1, G-21–22
  OCNs, F-3
  TPU, 559–560
Microbenchmarks
  disk array deconstruction, D-51–54
  disk deconstruction, D-48–50
Microinstructions, complications, C-45
MicroMIPS64, K-3
  16-bit instruction, **K-7**, **K-10**, **K-17**
  register encodings, **K-7**
Micro-op decode, 253
Micro-op fusion, ALU, 254
Microprocessor
  AMD Opteron, **27**
  clock rate, **26**
  design for typical case, 27
  do nothing well, 27
  energy and power within, 25–28
  growth rate, 2, **3**
  inside disks, D-4
  performance milestones, **22**
  recent advances, M-35
  VAX 11/780, **3**
Microprocessor without Interlocked Pipeline Stages (MIPS)
  early pipelined CPUs, M-28
  multiple-issue processor, M-30
  performance measurement history, M-6–7
  RISC history, M-20
Microsoft
  availability zones, **499**
  containers, M-76
Microsoft Azure, M-75–76
Microsoft Catapult
  Bing search engine, 573
  board design, **568**
  CNNs on, 570–572, **571–572**
  evaluating, 601–602
  guidelines, 577–579
  implementation and architecture, 568–569
  search acceleration on, 573–574
  software, 569
  version 1 deployment, 574
  version 2 deployment, 575–577, **576–577**
Microsoft's DirectX 8, M-51–52
Microsoft Windows, RAID benchmarks, **D-22**
Microsoft XBox, M-51–52
Migration, 379
Million instructions per second (MIPS)
  conditional instructions, H-27
  embedded systems, E-15
  MIPS16, **K-4**, K-5
  MIPS M2000, M-22, **M-22**
  MIPS M2000 *vs.* VAX 8700, **M-22**
  MIPS64 R6, K-19, **K-19**
  MIPS R2000, M-21
  MIPS R10000, 423
  MIPS R4000 pipeline, C-55–64
    floating-point operations, C-60–61, **C-60**
    performance, C-61–64
  RISC instruction set, C-3–4
  Sony PlayStation 2 Emotion Engine, E-17

Millions of floating-point operations per second (MFLOPS)
  early performance measures, M-7
  parallel processing debates, M-58–59
  SIMD computer history, M-56
  SIMD supercomputer development, M-46
  vector performance measures, G-15–16
Minibatches, DNNs, 556
Minicomputers, 4
MIPS. *See* Million instructions per second (MIPS)
MIPS R3000
  integer division/remainder, J-11–12
  integer overflow, **J-11**
MIPS R3010
  arithmetic functions, J-57–61
  chip comparison, **J-58**
  chip layout, **J-59–60**
  floating-point exceptions, J-35
MIPS R4000, early pipelined CPUs, M-28
Misprediction rate
  ARM Cortex-A53, **250**
  branch-prediction buffers, **C-25**
  profile-based predictor, **C-23**
  SPEC89 benchmark, **C-26**
  SPEC89 *vs.* predictor size, **187**
  static branch prediction, C-22, **C-23**
  tagged hybrid *vs.* gshare predictors, **190**
Misses per instruction
  advantage, B-6
  application/OS statistics, **B-59**
  and block size, 397–399, **398**
  cache performance, B-5–6
  memory hierarchy design, 82
Miss penalty, B-20–21
  cache optimization, B-30–36, 105–106, **106**
  compiler-controlled prefetching, 111–114
  critical word first, 104–105
  early restart, 104–105
  hardware prefetching, 109–111
  memory hierarchy design, 82
  multilevel caches reducing, 83
  reducing, 95
  reduction via multilevel caches, B-30–35
  write buffers, 83
Miss rate
  AMD Opteron example, B-15
  average memory access time, B-29–30, **B-30**
  bigger caches reducing, 83
  cache optimization
    and associativity, B-28–30
    and block size, B-26–28, **B-27**
    and cache size, **B-24–25**, B-28, **B-33**, **B-37**
    and virtually addressed cache size, **B-37**
  cache performance, B-16–17
  and cache size, **B-24–25**, B-28, **B-33**, **B-37**
  compiler-controlled prefetching, 111–114
  data, B-16
  and data cache size, **402**
  early IBM computers, M-11
  example calculations, B-6
  formula, B-16
  global, B-31
  hardware prefetching, 109–111
  higher associativity reducing, 83
  instruction, B-16
  larger blocks reducing, 82
  L1 caches, **402**
  L3 caches, 397–399, **398**
  local, B-31
  measurement, B-4–5
  memory hierarchy, 81
  memory stall clock cycles, B-4
  reducing, 95
  scientific workloads
    distributed-memory multiprocessors, **I-28–32**
    symmetric shared-memory multiprocessors, I-22, **I-23–25**
  total and distribution, **B-25**
  unified, B-16
Miss Status Handling Registers (MSHRs), 104
MIT Raw, **F-78**
Mixed cache, B-15
Mixer, radio receiver, **E-23**
M/M/1 model, D-30, D-32, D-57
M/M/2 model, D-57
Modified, shared, invalid (MSI) protocol, 388
Modified state, 383–384, 406
  large-scale multiprocessor cache coherence, I-35–36
Modula-3, integer division/remainder, **J-12**
Module availability, 37
Module reliability, 37
MOESI, 388
Moore's Law, 19
  interconnection networks, F-74
  pitfall, 58
  point-to-point links and switches, D-34
  RISC history, M-23
  semiconductor manufacturing, 4
  switch size, F-29
  transistors, 5, 540
Motion JPEG encoder, Sanyo VPC-SX500 digital camera, E-19
Motorola 68882, floating-point precisions, J-33
Motorola 68000, memory protection, M-10
Move address, K-55–56
MPEG
  multimedia SIMD extensions history, M-50
  Sanyo VPC-SX500 digital camera, E-19
  Sony PlayStation 2 Emotion Engine, E-17
MPPs. *See* Massively parallel processors (MPPs)
MTTF. *See* Mean time to failure (MTTF)
MTTR. *See* Mean time to repair (MTTR)
Multibanked caches, 99–100
Multichip modules, OCNs, F-3
Multicomputers, 370
  cluster history, M-65
  definition, M-59
  historical background, M-64–65
Multicore processor, 17, 369, 371–372, 382, 408
  approaches, 389

  architecture, **430**
  coherence, 387
  Cray X1E, G-24
  development, 404
  DSM, **373**, **405**, **452**
  Intel i7 performance and energy efficiency, 434–437
  on multiprogrammed workload, 426–432
  OCN, F-101
  performance, 426–437, **432**
  point-to-point, **446**
  scaling, **432**, 442–444
  single chip, 382, **391**, 446–451
  and SMT, 436–437
Multics protection software, M-9–10
Multicycle operations, RISC pipeline, C-45–49
  FP pipeline performance, C-55
  hazards and forwarding, C-49–52
  maintaining precise exceptions, C-53–55
Multidimensional arrays, 299–301
Multiflow processor, M-31–33
Multigrid methods, ocean application, I-9–10
Multilayer perceptrons (MLPs), 549–550
Multilevel caches
  centralized shared-memory architectures, 377
  memory hierarchy history, M-12
  miss penalty reduction, B-30–35, **B-33**
  write process, B-11
Multilevel exclusion, B-35
Multilevel inclusion, B-34, 423
  memory hierarchy history, M-12
Multimedia applications, desktop processor support, **E-11**
Multimedia Extensions (MMX), K-31, M-49–50
Multimedia instruction sets, 10
Multimedia SIMD extensions, M-49–50
  DSPs, E-11
Multimode fiber, interconnection networks, F-9–10
Multipass array multiplier, **J-51**
Multiple instruction streams, multiple data streams (MIMD), 11, 282, 369–370, 438–439
  early computers, M-57
  first vector computers, M-49
  multimedia SIMD and GPU *vs.*, 347–353
Multiple instruction streams, single data stream (MISD), 11
Multiple-issue processors
  advantages, 221–222
  challenges, 182, 221–222
  characteristics, **219**
  dynamically scheduled processor, 222, 224
  early development, M-29–32
  EPIC approach, 221
  instruction-level parallelism, 218–227
  microarchitectural techniques case study, 266–273
  with speculation, **223**
  superscalar, 218, 223
  VLIW approach, 218–222, **220**
Multiple lanes technique
  vector architecture, 293–294, **294**
  vector performance, G-7–9
Multiple-precision addition, J-13
Multiply-accumulate (MAC), 589
  DSP, E-5
  TI TMS320C55 DSP, E-8
Multiply operations
  chip comparison, J-61
  floating point
    denormals, J-20–21
    examples, **J-19**
    multiplication, J-17–21
    overview, J-17–20
    precision, J-21
    rounding, **J-18**, J-19
  integer arithmetic
    array multiplier, **J-50**
    Booth recoding, **J-49**
    even/odd array, **J-52**
    issues, J-11
    many adders, J-50–54, **J-50**
    multipass array multiplier, **J-51**
    n-bit unsigned integers, **J-4**
    Radix-2, J-4–7
    signed-digit addition table, **J-54**
    with single adder, J-47–49, **J-48–49**
    Wallace tree, **J-53**
  integer shifting over zeros, J-45
Multiprocessor
  application, 373–374
  architecture issues and approach, 370–373
  bus-based coherent, M-59–60
  cache coherence, 377–379
  cluster history, M-62–65
  coining of term, M-59
  definition, 369
  early computers, M-57
  early machines, M-57
  embedded systems, E-14–15
  execution time, 438
  factors, 368
  fallacy, 60
  Intel, 5
  large-scale (*see* Large-scale multiprocessors)
  linear speedup, 438–439, **440**
  parallel processing challenges, 373–377
  parallel processing debates, M-57–59
  performance gains, 424–426
  processor performance, 371–372
  recent advances and developments, M-59–60
  shared-memory (*see* Shared-memory multiprocessors (SMPs))
  SIMD computers, M-55–57
  synchronization and consistency models, M-64–65
  Xeon E7 MP scalability, 433–434
Multiprogramming, 369
  virtual memory, B-49, 119
  workload, 399–404, 426–432
Multistage interconnection networks (MINs)
  bidirectional, F-34
  crossbar switch calculations, **F-32–33**
  vs. direct network costs, F-96
  topology, F-32
Multistage switch fabrics, F-31
Multi-Streaming Processor (MSP)
  Cray X1, G-21–24, **G-22**
  Cray X1E, G-24
  first vector computers, M-49
Multithreaded SIMD Processor, 311, 315, **317**

Multithreading, 369
  hardware approaches, 243–244
  historical background, M-35–36
  instruction-level parallelism, 242–247
  parallel benchmarks, **247**
  performance gains, 424–426
  simultaneous (*see* Simultaneous multithreading (SMT))
  speedup from, **248**
  superscalar processors, 245–247
MVAPICH, F-81, **F-81**
M-way set associative, B-8
MXP processor, components, E-14–15
Myrinet SAN, F-49, M-63, M-74, F-90, **F-90**

N

NAK. *See* Negative acknowledge (NAK)
Name dependences
  instruction-level parallelism, 172–173
  register renaming, 196
Nameplate power rating, 482
NaN. *See* Not a Number (NaN)
NASA Ames Research Center, M-46, M-65
NAS parallel benchmarks, vector processor history, G-27–28
National Science Foundation, F-102
Natural parallelism embedded systems, E-15
n-bit adder, carry-lookahead, **J-38**
n-bit number representation, J-7–10
n-bit unsigned integers division, **J-4**
N-body algorithms, Barnes application, I-8–9
NBS DYSEAC, M-89
N-cube topology, F-36–38
NEC SX/2, M-48
NEC SX/5, M-48–49
NEC SX/6, M-48–49
NEC SX-8, M-48–49
NEC SX-9, M-49
  first vector computers, M-49
NEC VR 4122, embedded benchmarks, E-13
Negative acknowledge (NAK)
  cache coherence, I-39
  directory controller, I-41
  DSM multiprocessor cache coherence, I-37
Negative-first routing, F-48
Nested page tables, 146
Netscape, F-102
Network Appliance (NetApp), D-9, D-41–43
Network-attached storage (NAS), M-88, 478
Network bandwidth, interconnection network, F-18
Network-Based Computer Laboratory (Ohio State), F-80–81
Network buffers, network interfaces, F-8
Network fabric, F-24–25
Network File System (NFS)
  benchmarking, D-20, **D-20**
  block servers *vs.* filers, D-35
  interconnection networks, F-93–94
  TCP/IP, F-86
Networking
  Google WSCs, 510–511
  performance milestones, **22**
Network injection bandwidth
  interconnection network, F-19–20
  multi-device interconnection networks, F-26
Network interface
  fault tolerance, F-67
  functions, F-6
  message composition/processing, F-6–9
Network interface card (NIC)
  functions, F-8
  I/O subsystem, F-95
  vs. switches, F-90, **F-90**
  zero-copy protocols, F-95
Network I/O, 467
Network layer, **F-84**
Network nodes, F-23, F-35, **F-36**
  distributed switched networks, F-35
Network of Workstations, M-63, M-74–75
Network on chip (NoC), F-3
Network ports, F-30
Network protocol layer, F-10
Network reception bandwidth, F-19–20
Network reconfiguration, F-70–71
Network technology, 20
  personal computers, F-2
Newton's iteration, J-27–30, **J-28**
Nicely, Thomas, J-65
Nodes
  communication bandwidth, I-3
  direct network topology, **F-37**
  distributed switched networks, F-35–40
  IBM Blue Gene/L, I-42–44, **I-43**
  IBM Blue Gene/L 3D torus network, F-76
  network topology performance and costs, **F-40**
  points-to analysis, H-9
Nokia cell phone, circuit board, E-24, **E-24**
Nonatomic operations, cache coherence, 386
Nonbinding prefetch, 111
Nonblocking caches
  cache bandwidth, 100–104
  case study, 148–164
  effectiveness, **102**
  implementing, 103–104
  memory hierarchy history, M-11–12
Nonblocking crossbar, F-33
Nonfaulting prefetches, 111
Nonoptimal replacement algorithm, M-11
Non-overlapped latency, B-20
Nonrecurring engineering (NRE) costs, 542
Nonrestoring division, J-5, **J-6**
Nonuniform cache access (NUCA), 371, 390, 426
Nonuniform memory access (NUMA), 372–373, 391, 426–429
  large-scale multiprocessors history, M-61
Non-unit strides, 300
  vector processor, G-25
North-last routing, F-48
Not a Number (NaN), J-14–16, J-21, J-34–35
Notifications, interconnection networks, F-10
NOW project, M-74–75
No-write allocate, B-11–12
NSFNET, F-102

NTSC/PAL encoder, Sanyo VPC-SX500 digital camera, E-19
NUMA. *See* Nonuniform memory access (NUMA)
NVIDIA GeForce, M-51–52
NVIDIA K80 GPU die Roofline, **599**
NVIDIA system
  computational structures, 313–320
  GPU computational structures, 313–320
  GPU computing history, M-52–53
  GPU instruction set architecture, 320–323
  GPU memory structures, 326–328, **327**
  graphics pipeline history, M-51–52
  instruction set architecture, 320–323
  NVIDIA P100, **354**
  scalable GPUs, M-51
  Tegra Parker system *vs.* Core i7, 346–353
N-way set associative, B-8
  conflict misses, B-23
  memory hierarchy, 81
  TLB, B-49
NYU Ultracomputer, M-61

O

Occupancy, communication bandwidth, I-3
Ocean application
  characteristics, I-9–10
  distributed-memory multiprocessor, I-30, **I-32**
  example calculations, I-11–12
  miss rates, **I-28**
  symmetric shared-memory multiprocessors, **I-23**
Offline reconstruction, RAID, D-55
Offload engines
  network interfaces, F-8
  TCP/IP reliance, F-100
Offset, B-8–9
  address, B-55–56
  block identification, B-8–9
  cache optimization, B-38
  destination, IA-32 segment, **B-53**
  IA-32 segment, **B-53**
  Opteron data cache, **B-13**, B-14
  page, B-38
  sign-extended, C-5
  virtual memory, B-43–44, B-46, B-55–56
  word, C-28
OLTP. *See* Online transaction processing (OLTP)
Omega, **F-31**, F-32–33
On-chip memory, embedded systems, E-4
On-chip networks (OCNs)
  basic considerations, F-3
  commercial implementations, F-73–74
  commercial interconnection networks, F-63
  cross-company interoperability, F-68
  effective bandwidth, F-19
  interconnection network domain relationship, F-4, **F-5**
  latency *vs.* nodes, **F-28**
  packet latency, **F-13**, F-14–16
  time of flight, F-14
  topology, F-30
  wormhole switching, F-52
One's complement, J-7–8
One-way set associativity, conflict misses, B-23
Online reconstruction, RAID, D-55
Online transaction processing (OLTP), 44, 395–396, **396**, 401
  storage system benchmarks, D-18
Opcode, A-21
OpenGL, M-51–53
Open instruction set, DSAs, 594
Open Systems Interconnect (OSI)
  Ethernet, F-86
  layers, **F-84**
Operand delivery stage, Itanium 2, H-42
Operands
  DSP, E-6
  instruction set architecture, **A-4**, A-13–15
  read, C-68
    dynamic scheduling pipelines, C-66
    ID pipe stage, 194
  TMS320C55 DSP, **E-6**
  type and size, A-13–15
Operating systems (general)
  address translation, B-38
  communication performance, F-8
  disk access scheduling, D-44
  memory protection performance, B-58
  miss statistics, **B-59**
  and page size, B-58
  segmented virtual memory, B-54
  storage systems, D-35–36
  vendor-independent, 2
  workload, 399–404
Operational costs count, 468
Operational expenditures (OPEX), 36, 486–490, **488**
Operation faults, D-11
Operations, 14 *See also specific types of operations*
  atomic, 386
  in instruction set, A-15–16, **A-15**
Operator dependability, disks, D-13–15
Opteron Data Cache, B-11
Optical media, interconnection networks, F-9–10
Oracle database, miss statistics, **B-59**
Ordering, and deadlock, F-47–48
Organizations, 17
  block placement, B-7–8, **B-7**
  buffer, F-59–61
  data dependences, 172
  designing, 17–18
  dynamic random-access memory (DRAM), **86**
  on-chip
    IBM Power8, **428**
    Xeon E7, **428**
  Opteron data cache, B-12–15, **B-13**
  Sony PlayStation Emotion Engine, **E-18**
  SPEC benchmarks, 433–434
Out-of-order completion
  definition, C-53
  dynamic scheduling pipelines, C-66
Out-of-order execution
  and cache miss, B-2
  dynamic scheduling, 193–194
  dynamic scheduling pipelines, C-66
  memory hierarchy, B-2
  microarchitectural techniques case study, 266–273

- Out-of-order execution *(Continued)*
  - miss penalty, B-20–21
  - Tomasulo's scheme, 208
- Out-of-order processors, memory hierarchy history, M-12
- Output buffered switch, F-56, **F-58**, F-61, **F-65**
- Output dependence, 173
  - compiler history, M-32
  - finding, H-7–8
- Output gate, 553
- Overclocking, 28
- Overflow, integer arithmetic, J-8, J-10–11, **J-11**
- Overhead
  - Amdahl's law, F-96
  - calculating, **F-94**
  - communication latency, I-4
  - interconnection networks, **F-27–29**
  - OCNs *vs.* SANs, **F-28**
  - processor, **G-4**
  - software, F-96
  - sorting case study, D-64–67
  - time of flight, F-14
- Overlapping triplets, integer multiplication, J-49
- Oversubscription, 478

P

- Packed decimal, A-14
- Packets
  - ATM, **F-79**
  - bidirectional rings, F-36
  - centralized switched networks, F-33
  - discarding, F-69
  - effective bandwidth *vs.* packet size, **F-19**
  - format example, **F-7**
  - InfiniBand, **F-79**
  - latency issues, F-13, **F-13**
  - lossless *vs.* lossy networks, F-12
  - network interfaces, F-8
  - network routing, F-22
  - switching, F-51
  - switch microarchitecture, F-56–59
    - pipelining, F-64–66
  - TI TMS320C6x DSP, **E-10**
  - topology, F-21–22
  - transport, interconnection networks, F-9–12
- Page(s)
  - coloring, B-38
  - definition, B-43
  - *vs.* segments, **B-43**
- Paged segments, B-43–44, **B-43**
- Paged virtual memory, B-54–57
- Page fault
  - definition, B-2–3, B-42, B-45
  - exception stopping/restarting, C-41
- Page offset
  - definition, B-38
  - virtual memory, B-43–44, B-46
- Page size, B-56
  - operating systems and, B-58
  - selection, B-46–47
  - virtual memory, B-46–47
- Page table entry (PTE)
  - definition, B-44, 136
  - fields in, B-52
- Page tables
  - AMD64 paged virtual memory, B-55
  - descriptor tables as, B-52
  - main memory block, B-44–45
  - nested, 146
  - protection processes, B-50
  - segmented virtual memory, B-51–52
  - shadow, 123
  - size, B-47
  - virtual address to physical address, **B-45**
- Paging support, 330
- Paired single operations, DSP media extensions, E-10–11
- Palt, definition, B-2–3
- Papadopolous, Greg, M-76
- Parallel architectures, classes of, 10–11
- Parallelism
  - challenges, 373–377
  - classes of, 10–11
  - computer design principles, 48
  - dependence analysis, H-8–9
  - Ethernet, F-74, F-82–83
  - exploitation statically, H-2
  - exposing with hardware support, H-23–27
  - global code scheduling, H-15–23, **H-16**
  - IA-64 instruction format, H-34–37
  - ILP (*see* Instruction-level parallelism (ILP))
  - request-level, 369
  - software pipelining, H-12–15
  - superblock scheduling, H-21–23, **H-22**
  - taking advantage of, 48
  - thread-level (*see* Thread-level parallelism (TLP))
  - trace scheduling, H-19–21, **H-20**
- Parallel memory systems, highly, 150–153
- Parallel processing, 369
- Parallel processors
  - areas of debate, M-57–59
  - bus-based coherent multiprocessors, M-59–60
  - cluster history, M-62–65
  - large-scale multiprocessors history, M-60–62
  - recent advances and developments, M-59–60
  - scientific applications, I-33–34
  - SIMD computers history, M-55–57
  - synchronization and consistency models, M-64–65
  - virtual memory history, M-65
- Parallel programming
  - computation communication, I-10–12
  - with large-scale multiprocessors, I-2
- Parallel Thread Execution (PTX), 320–323, **322**
- Paravirtualization, 126
- PA-RISC, K-3
- Parity, dirty bits, D-61–64
- PARSEC benchmark
  - simultaneous multithreading, 246
  - without SMT, 435–437, **435**, **437**
- Partial disk failure, dirty bits, D-61–64
- Partial store order (PSO), 420, **421**, 457
- Partitioned add operation, DSP media extensions, E-10
- Partitioning, 480–482
- Pascal GPU architecture, 328–331, **329**
  - full-chip block diagram, **318**
  - SIMD Processor, **330**
- Pascal programs, integer division/remainder, **J-12**

Pattern, disk array deconstruction, D-51
Payload
  messages, F-6
  packet format, **F-7**
p-bits, J-21–23, J-25, J-36–37
PC. *See* Program counter (PC)
PCI bus, historical background, M-88
PCI-Express (PCIe), storage area network, F-29
PCI-X, M-88
  storage area network, F-29
PCI-X 2.0, F-67
PCMCIA slot, Sony PlayStation 2 Emotion Engine case study, E-15
PC-relative addressing, A-9, K-52
PDP-11, M-10–11, M-19, M-57, M-88
Peak performance
  Cray X1E, G-24
  DAXPY on VMIPS, G-21
  fallacy, 63, **63**
  in vector architectures, 355
  VMIPS on DAXPY, G-17
Peer-to-peer, wireless networks, E-22
Pegasus, M-17
PennySort competition, D-66
Pentium, K-31–32
Perfect Club benchmarks
  vectorization, 303, **303**
  vector processor history, G-27–28
Perfect-shuffle exchange, F-32
Performability, RAID reconstruction, D-55–57
Performance. *See also* Cost-performance; Peak performance
  benchmarks, 40–45
  branch scheme, C-21–22, **C-22**
  cache (*see* Cache performance)
  common data bus, 207
  deep neural networks, 603
  dirty bits, D-61–64
  disk array deconstruction, D-51–54
  disk deconstruction, D-48–50
  DSAs, 600–601, **601**
  embedded computers, **E-13–14**
  Fujitsu SPARC64 X+, 429–431, **432**
  IBM Power8, 431–432, **432**
  instruction set architecture, 258
  Intel Core i7 920, 434–437
  Intel Core i7 6700, 138–142, 255–257
  Internet Archive Cluster, D-36–41
  interprocessor communication, I-3
  I/O devices, D-15–23
  I/O subsystem design, D-59–61
  I/O system design/evaluation, D-36–41
  Itanium 2, H-43, **H-43**
  large-scale multiprocessors, scientific application
    distributed-memory multiprocessors, I-26–32, **I-28–32**
    parallel processors, I-33–34
    symmetric shared-memory multiprocessor, I-21–26, **I-23–26**
    synchronization, I-12–16
  latency, 20, **21**
  measuring, 39–47
  microprocessor, **22**
  multicore processor, 426–437, **432**
  quantitative measures, M-6–7
  reporting, 39–47
  sorting case study, D-64–67
  summarizing, 39–47
  symmetric shared-memory multiprocessor, I-21–26, **I-23–26**
  trends, 20
  vector processor, G-2–9
    chaining, G-11–12, **G-12**
    DAXPY on VMIPS, G-19–21
    sparse matrices, G-12–14
    start-up and multiple lanes, G-7–9
    unchaining, **G-12**
  VMIPS on Linpack, G-17–19
Permanent failure, commercial interconnection networks, F-70
Permanent faults, D-11, 93
Personal computers, 4, 6
  LANs, F-4
  networks, F-2
  PCIe, F-67
Personal mobile device (PMD), A-2, 7–8
  image-processing unit for, 542
  memory hierarchy in, **79**
PetaBox GB2000, Internet Archive Cluster, D-37
Phase-change memory (PCM)
  memory hierarchy design, 93
  Xpoint memory chips, 93
Phase-ordering problem, A-26
Physical address
  AMD Opteron data cache, B-12–13
  AMD64 paged virtual memory, B-55
  safe calls, B-54
  translation, B-36–40
  virtual address, **B-45**
  virtual memory, B-42, B-51
Physical cache, B-36–37
Physical channels, F-47–48
Physical layer, **F-84**
Physical memory
  centralized shared-memory multiprocessor, **372**
  directory-based cache coherence, 380
  memory hierarchy, B-40–42
  virtual machine, 123
Physical register
  instruction, **237**
  register renaming, 235
  SIMD instructions, 320
  uses of, 234–235
Physical transfer units (phits), F-64
Physical volumes, D-35
PicoJoules, **541**
PID. *See* Process-identifier tag (PID)
Pin-out bandwidth, topology, F-39
Pipelined circuit switching, F-51
Pipelined CPUs, early versions, M-27–28
Pipeline delays, C-44
Pipeline interlock, C-17
Pipeline latches, C-30–31
Pipeline organization, data dependences, 172
Pipeline registers
  data hazards stall minimization, C-14
  definition, C-30–32
  pipelining performance, C-8–10
Pipeline scheduling
  instruction-level parallelism, 177–182
  microarchitectural techniques case study, 266–273

- Pipeline stall cycles
  - ARM Cortex-A53, 250–251
  - branch scheme performance, C-21
- Pipelining
  - branch cost reduction, C-22
  - branch hazards, C-18–22
  - branch issues, C-35–37
  - branch penalty reduction, C-19–20
  - branch-prediction buffers, C-23–25, **C-24–26**
  - branch scheme performance, C-21–22, **C-22**
  - cache access, 99–100
  - classic stages for RISC processor, C-6–8, **C-7**
  - compiler scheduling, M-32
  - computer design principles, 48
  - concept, C-2–3
  - data hazards, C-12–17
    - definition, C-11
    - instruction set complications, C-45
    - pipelined execution of instructions, C-13
    - stall minimization by forwarding, C-14–15, **C-15–16**
    - stall requirements, C-16–17
    - types, C-12
  - definition, C-2
  - detection of hazard, **C-34**
  - example, **C-7**
  - exception
    - arithmetic-logical units, C-5
    - categories, **C-40**
    - floating-point, C-41–42
    - precise, C-41–44
    - RISC V, C-42–43, **C-42**
    - stopping/restarting, C-41–42
    - types and requirements, C-38–41, **C-40**
    - unexpected sequences, C-70
  - floating-point addition speedup, J-25
  - graphics pipeline history, M-51–52
  - hazards, C-10–25
  - instruction set complications, C-43–45
  - interconnection networks, F-12
  - MIPS R4000, C-55–64
  - performance issues, C-8–10
  - performance with stalls, C-11–12
  - predicted-not-taken scheme, C-19–20, **C-19**
  - RISC V, C-30–33
    - classic pipeline stages, C-6–8
    - control, C-33–35
    - exception, C-42–43, **C-42**
    - FP pipeline, C-45–55, **C-57**
    - instruction set, C-3–4, C-65
    - instruction set complications, C-43–45
    - integer pipeline to handle multicycle operations, C-45–55
    - multicycle FP operations, C-45–55
    - pipeline control, C-33–35
    - simple implementation, C-4–6, **C-6**, C-26–29, **C-30**
  - simple implementation, C-26–37
  - speedup from, C-11–12
  - static branch prediction, C-22, **C-23**
  - switch microarchitecture, F-64–66
- Pipe segment, C-3
- Pipe stage
  - definition, C-3
  - exception stopping/restarting, C-41
  - performance issues, C-8–10
  - program counter, **C-31–32**
  - register additions, **C-31**
  - RISC processor, C-8
- Pixel Visual Core
  - architecture philosophy, 583–584
  - evaluating, 601–602
  - example, 588
  - floor plan, **592**
  - Halo, 584–585
  - implementation, 590–591
  - instruction set architecture, 587–588
  - line buffers in, 590
  - processing element, 588–589
  - processor, 585–587
  - programmer view of, **589**
  - software, 582
  - two-dimensional array, **586**
  - two-dimensional line buffers, 589–590
- PLA, early computer arithmetic, J-65
- Points-to analysis, H-9
- Point-to-point links
  - bus replacement, **D-34**
  - Ethernet, F-30
  - storage systems, D-34
  - switched-media networks, F-24–25
- Poison bits, compiler-based speculation, H-28, H-30–31
- Poisson distribution
  - basic equation, D-28
  - random variables, D-26–34
- Poisson, Siméon, D-28
- Polycyclic scheduling, M-32
- Portable computers, interconnection networks, F-89
- Portable mobile devices (PMDs), 580–581
- Port number, network interfaces, F-7–8
- Position independence, A-17
- Power, 442–443
  - first-level caches, 95–98
  - within microprocessor, 25–28
  - systems perspective, 23–24
- Power 3, K-25
  - additional instructions, **K-24**
  - branch registers, K-23–25
- Power consumption. *See also* Energy efficiency
  - case study, 69–71
  - embedded benchmarks, E-13
  - interconnection networks, F-61, F-89
  - simultaneous multithreading, 246
  - speculation, 238–239
  - TI TMS320C55 DSP, E-8
- Power gating, 28
- PowerPC, K-6, K-11, K-25
  - cluster history, M-64
  - conditional instructions, H-27
  - IBM Blue Gene/L, I-41–42
  - RISC history, M-21
- PowerPC AltiVec, multimedia support, **E-11**
- Power-performance
  - Dell PowerEdge servers, 55–58, **56**
  - of servers, **57**
- Power utilization effectiveness (PUE)
  - pitfall, 515
  - WSCs, 483, **484–485**
- Precise exceptions
  - definition, C-41–44
  - maintaining, C-53–55

Precisions, floating-point arithmetic, J-33–34
Predicated instructions
  example calculations, H-25
  exposing parallelism, H-23–27
  IA-64, H-38–40
Predicate registers, 296–298
  IA-64, H-34
Predication, TI TMS320C6x DSP, E-10
Predicted-not-taken scheme, C-19–20, **C-19**
Prediction. *See also* Misprediction rate
  branch
    accuracy, **C-25–26**
    cost reduction, C-22
    dynamic, C-23–25
    instruction-level parallelism, 182–191
    static, C-22, **C-23**
  branch-prediction buffers, C-23–25, **C-24–26**
  return address, 232–234
  size, **187**
  2-bit scheme, **C-24**, 182, 184, **185**
Prediction by Partial Matching (PPM), 188
Prefetching, 376
  instruction, 234
  Itanium 2, H-42
  memory hierarchy design, 138
  software and hardware, 148–164
Prefix, Intel 80x86 integer operations, K-35
Presentation layer, **F-84**
Present bit, B-52
Price, cost *vs.*, 35
Price-performance, 8
Primitives, synchronization, M-64–65
Principle of locality
  coining of term, M-11
  computer design principles, 48–49
  definition, B-2
  scientific workloads on symmetric shared-memory multiprocessors, I-25–26
Private data, 377
Private memory, NVIDIA GPU memory structures, 326
Procedure calls
  IA-64 register model, H-33–34
  VAX, K-57
Procedure invocation options, A-19–20
Process-complexity factor, 34
Process concept, B-49, 119
Process-identifier tag (PID), B-37–38, **B-37**
Processing element (PE) array, 570, **572**, 580, 584–585, 587, 592
Processor consistency, 420
Processor cycle
  cache performance, B-3
  definition, C-3
Processor-dependent optimizations, A-26
Processor-intensive benchmarks, 41
Processor performance
  average memory access time and, B-17–20
  equation, 52–55
  multiprocessors, 371–372
Process switch
  definition, **B-37**, B-49
  virtual memory, B-49, 119
Producer-server model, D-15–16, **D-16**
Profile-based predictor, misprediction rate, **C-23**
Program counter (PC), A-17
  branch hazards, C-18
  branch-target buffers, 228–229, **229**
  dynamic branch prediction, C-23–24
  exception stopping/restarting, C-41–42
  pipeline branch issues, C-35–36
  pipe stage, **C-31–32**
  precise exceptions, C-54
  RISC V instruction set, C-4
  simple RISC implementation, C-27–29
  tagged hybrid predictors, 188–189
Programmer's view, memory consistency, 418–419
Programming models, warehouse-scale computers, 471–476
Program order
  cache coherence, 378–379
  control dependence, 174–175
  data hazards, 173–174
  definition, 173
Protection schemes
  cross-cutting issues, 126–127
  development, M-9–12
  network interfaces, F-8
  Pentium *vs.* Opteron, B-57
  processes, B-50
  safe calls, B-54
  segmented virtual memory, B-51–54
  virtual memory, B-41
Protocol deadlock, routing, F-45–46
Protocol stack, F-86–87
PSO. *See* Partial store order (PSO)
PTE. *See* Page table entry (PTE)
Pulse amplitude modulation (PAM-4), 59

Q
QCDOD, M-64
QPI. *See* QuickPath Interconnect (QPI)
QsNetII, F-67, **F-80**
Quadrics SAN, **F-80**
Quality of service (QoS)
  dependability benchmarks, D-21
  WAN, F-102–103
Quantitative performance measures, development, M-6–7
Quantization, DNNs, 556
Queue
  definition, D-25
  discipline, D-26
  Intel Core i7, **256**
  waiting time calculations, D-26
Queuing locks, large-scale multiprocessor synchronization, I-18–21
Queuing theory, D-23–34
QuickPath Interconnect (QPI), 426–429

R
Race-to-halt, 28
Radio frequency amplifier, radio receiver, **E-23**
Radio receiver, components, **E-23**
Radio waves, wireless networks, E-21
Radix-8 multiplication, J-49
Radix-2 multiplication/division, J-4–7, **J-4**, **J-55**
Radix-4 multiplication/division, J-49, **J-49**, J-56, **J-56–58**, J-61

RAID. *See* Redundant array of inexpensive disks (RAID)
Random access memory (RAM), F-56
Random Access Method of Accounting Control (RAMAC), M-85, M-88–89
Random replacement, B-9–10, **B-10**
Random variables, distribution, D-26–34
Ranking, 573
Ransomware, 491
Ray casting (RC), 350
Read after write (RAW), C-12–14
  check for, C-52
  first vector computers, M-47–48
  instruction set complications, C-44
  program order, 173
  RISC pipeline control, C-34
  stalls, C-49, **C-50**, C-51
  TI TMS320C55 DSP, E-8
  Tomasulo's algorithm, 195, 217
Read miss, 382–383, 410
  cache coherence, **384–385**, 386, 388
  directory-based cache coherence protocol, 410–411
  memory stall clock cycles, B-4
  miss penalty reduction, B-35–36
  Opteron data cache, B-14
Read operand, C-68
  dynamic scheduling pipelines, C-66
  ID pipe stage, 194
Real addressing mode, K-31
Real memory, 123, 126
Real mode, **K-36**
Real numbers, K-39
Real-time constraints, E-2
Real-time performance, 7–8
  requirement, definition, E-3–4
Real-time processing, embedded systems, E-3–4
Rearrangeably nonblocking, F-33
Receiving overhead, F-28, **F-28**, F-41, F-67
Receiving overhead, communication latency, I-3
Reconfiguration deadlock, F-45–46
Recovery time, vector processor, G-8
Rectified linear unit (ReLU), 546
Recurrences
  basic approach, H-11
  loop-carried dependences, H-5
Recurrent neural network (RNN), 553–555
Red-black Gauss-Seidel, I-9–10
Reduced Instruction Set Computer (RISC), 413–414, 423
  cache performance, B-6
  compiler history, M-33
  correlating predictors, 183
  development, 2
  early pipelined CPUs, M-28
  FENCE in, 420–422
  historical background, M-20–23, **M-22**
  multimedia SIMD extensions history, M-50
  pipeline scheduling and loop unrolling, 177–178
  reduced code size in, A-23–24
  RISC-I, M-20–21
  RISC-II, M-20–21
  Sanyo VPC-SX500 digital camera, E-19
  vector processor history, G-26
Reduced Instruction Set Computer (RISC) architectures, A-33–42
  addressing modes and instruction formats, K-6–9
  ARM architecture, K-22
  16-bit instructions, K-3, **K-4**, K-5, **K-10**
  compare and conditional branch, K-11–16
  conditional branches, **K-17**
  control instructions, **K-18**
  data transfer instructions, **K-18**
  digital signal-processing extensions, K-28
  extensions beyond RV64G, K-18–19
  for lower-end applications, K-3
  MIPS64 R6, K-19
  multimedia and graphics operations, K-25–27
  Power3, K-23–25
  RISC-V integer ISA, **K-13**
  RV64GC core 16-bit instructions, K-16–17
  RV64G core instructions, K-11
  RV64G instruction, K-5, **K-10**, **K-12**
  SIMD extensions, K-25–27
  SPARC v.9, K-20–22
  survey of, K-3–29
Reduced Instruction Set Computer (RISC) V, 12–17, **13**, 413–414, 423
  addressing modes, A-36
  control flow instructions, A-39–40
  data types for, A-35–36
  dies, **33**
  FENCE in, 420–422
  floating point instructions for, **16**
  floating-point operations, A-40–41
  instruction format, A-36–37
  instruction set architecture formats, **16**
  instruction set organization, A-34
  load and store instructions, **A-38**
  operations, A-37–39
  pipelining, C-30–33
    classic pipeline stages, C-6–8
    control, C-33–35
    exception, C-42–43, **C-42**
    FP pipeline, C-45–55, **C-57**
    instruction set, C-3–4, C-65
    instruction set complications, C-43–45
    integer pipeline to handle multicycle operations, C-45–55
    multicycle FP operations, C-45–55
    pipeline control, C-33–35
    simple implementation, C-4–6, **C-6**, C-26–29, **C-30**
  registers for, A-34–35
  scalar architecture, RV64V, **284**
  SIMD Processor, 306–307
  SPECint2006 programs, **A-42**
  subset of instructions in, **15**
  Tomasulo's algorithm
    floating-point operation, **198**
    instruction set, 195
Reductions, 344–345
Redundancy
  chip fabrication cost case study, 67–68

- computer system power consumption case study, 69–71
- index checks, B-9
- simple RISC implementation, C-29

Redundant array of inexpensive disks (RAID)
- dependability benchmarks, D-21–23
- disk array deconstruction case study, D-51–54
- disk deconstruction case study, D-48–50
- hardware dependability, D-15
- historical background, M-86–88
- I/O subsystem design, D-59–61
- logical units, D-35
- NetApp FAS6000 filer, D-41–43
- overview, D-6–8
- performance prediction, D-57–59
- RAID 0, D-6
- RAID 1, D-6, M-87
- RAID 2, D-6, M-87
- RAID 3, D-6, M-87
- RAID 4, D-7, M-87
- RAID 5, D-8, M-87
- RAID 6, D-8–10
- RAID 10, D-8
- row-diagonal parity, D-9–10, **D-9**, D-41–42

Redundant multiplication, integers, J-47
Reference bit, B-45, B-52
Regional explicit congestion notification (RECN), F-70
Register(s)
- DSP examples, **E-6**
- IA-64, H-33–34
- instructions and hazards, **C-13**
- network interface functions, F-7
- pipe stage, **C-31**
- tag, **202**

Register addressing, K-52
Register allocation, A-26–27
Register deferred addressing, K-52–53
Register fetch (RF)
- cycle, C-5
- MIPS R4000 pipeline, C-56
- simple RISC implementation, C-27

Register file, C-27, 200–201
- data hazards, C-16–17
- floating-point operations, **C-50**
- OCNs, F-3
- precise exceptions, C-54
- RISC instruction set, C-5, C-7–8
- simple RISC implementation, C-29

Register indirect addressing mode, K-34
Register management software-pipelined loops, H-14
Register-memory
- architecture, A-3
- ISAs, 12

Register prefetch, 111
Register pressure, 182
Register renaming
- antidependence, 196
- deallocating registers, 235
- definition, 173, 195–196
- expected output, **269**
- initial state table, **270**
- microarchitectural techniques case study, 266–273
- name dependences, 196
- *vs.* reorder buffers, 234–236
- reservation stations, 196–197, 199–200
- sample code, **269–270**

Register stack engine, IA-64, H-34
Register Transfer Level (RTL) code, 569
Regularity, bidirectional MINs, F-33–34
Reinforcement learning (RL), 549
Relaxed consistency models, 419–422, **421**
Release consistency (RC), 420–422, **421**, 457
Reliability
- commercial interconnection networks, **F-37**
- I/O subsystem design, D-59–61
- storage systems, D-44

Relocation, virtual memory, B-41–42
Remainder, floating point, J-31–32
Remington-Rand, M-5
Remote direct memory access (RDMA), F-80
Remote node, 406–407
Renaming map, 235
Reorder buffer (ROB)
- compiler-based speculation, H-31–32
- hardware-based speculation, 209–212, 214–215
- issue with, 236
- register renaming *vs.*, 234–236

Replication
- definition, 377, 379
- virtual memory, B-49

Reply, messages, F-6
Reproducibility, 45
Request
- messages, F-6
- switch microarchitecture, F-58–59

Requested protection level, B-54
Request-level parallelism (RLP)
- definition, 5, 10–11, 369
- WSCs, 467

Request phase, F-49–50
Request-reply deadlock, F-45–46
Reservation stations
- common data bus, **202**
- fields, 199–200
- register renaming, 196–197, 199–200

Reserved register, 414
Resource sparing, F-70–71
Response time. *See also* Latency
- definition, 20, 39
- DNN applications, 596–600
- I/O benchmarks, **D-18**
- producer-server model, **D-16**
- *vs.* throughput, D-16–18, **D-17**

Restartable pipeline
- definition, C-40–41
- exception, C-41–42

Restorations, dependability, 37
Restoring division, J-5, **J-6**
Resume event, exception, C-40
Return address, predictors, 232–234
Returns, cache coherence, 378–379
Reverse path, cell phones, E-24
RF. *See* Register fetch (RF)
Rings, F-43
- protection processes, B-50

Ripple-carry adder, J-2–3, **J-3**
- carry-lookahead adder with, **J-42**
- chip comparison, J-61

Ripply-carry addition, J-2–3

RISC. *See* Reduced Instruction Set Computer (RISC)
ROB. *See* Reorder buffer (ROB)
Role code, 569, **570**
Roofline model, **349**
  CPUs *vs.* GPUs, **355**
  DNN applications, 596–600, **597**
Round digit, J-18
Rounding modes, J-14, J-17–20, **J-18**, **J-20**
  FP precisions, J-34
  fused multiply-add, J-33
Round-robin (RR), F-49
Routers, **F-64**, F-83
Routing algorithm, F-21–22, F-45–49
Row access strobe (RAS), memory hierarchy design, 85–86
Row-diagonal parity, D-9–10, **D-9**, D-41–42
Row major order, blocking, 107
RV64c, 16-bit instruction formats, **K-7**
RV32E, **K-4**
RV64GC, K-3
  ALU instructions in, **K-17**
  16-bit instructions, **K-10**
  core 16-bit instructions, K-16–17
  register encodings, **K-7**
RV64G core instructions, K-11
RV64G, extensions beyond, K-18–19
RV64G instruction, **K-12**
RV64V extension
  data sizes, **287**
  vector architecture, 283–287, **284**
  vector instructions, **286**
RV64V instruction set, 293

S
Sanyo digital cameras, SOC, **E-20**
Sanyo VPC-SX500 digital camera, embedded system case study, E-19
SASI, M-88
SATA disks. *See* Serial Advanced Technology Attachment (SATA) disks
Saturating arithmetic, DSP media extensions, E-11
Scalability
  computer design principles, 48
  server systems, 9
Scalable GPUs, M-51
Scalar lane (SCL), 586–588
Scalar processors, 310, 326, **332**, 334. *See also* Superscalar processors
  early pipelined CPUs, M-28
  vs. vector, G-19
Scalar registers
  Cray X1, G-21–22
  set of, 285
Scaled speedup. *See* Weak scaling
Scaling
  CMOS, 442–443
  computation-to-communication ratios, **I-11**
  instruction-level parallelism, 442
  multicore processor, **432**, 442–444
  scientific applications on parallel processing, I-34
  SPECintRate benchmarks, 429–431, **431**
  strong, 439
  weak, 439
Scan Line Interleave (SLI), M-51
Scatter store, 301–302
Schorr, Herb, M-29–30
Scientific applications
  Barnes, I-8–9
  characteristics, I-6–12
  cluster history, M-62–63
  distributed-memory multiprocessors, I-26–32, **I-28–32**
  FFT kernel, I-7
  LU kernel, I-8
  ocean, I-9–10
  parallel processors, I-33–34
  parallel program computation/communication, I-10–12, **I-11**
  parallel programming, I-2
  symmetric shared-memory multiprocessor, I-21–26, **I-23–26**
Scoreboarding
  definition, 194–195
  dynamic scheduling with, C-66–70, **C-68**
SCSI. *See* Small Computer System Interface (SCSI)
SDRWAVE, J-62
Second-level caches
  cache optimization, B-30–35, **B-34**
  execution time, B-32, **B-34**
  interconnection network, F-74
  Itanium 2, **H-41**
  memory hierarchy, B-48–49, **B-48**
  miss rate calculations, B-30–35, **B-34**
Secure Virtual Machine (SVM), 146
Seek distance, D-46, **D-46–47**
Seek time, storage disks, D-45–46, **D-46**
Segment descriptor, B-52, **B-53**
Segmented virtual memory
  bounds checking, B-52
  Intel Pentium processors, B-51–54
  memory mapping, B-52
  safe calls, B-54
  sharing and protection, B-52–53
Segments
  definition, B-43
  pages *vs.*, **B-43**
Self-correction, Newton's algorithm, J-28–30
Self-draining pipelines, M-30
Self-routing, MINs, F-48–49
Semiconductor
  DRAM, 19
  flash, 19
  ITRS, 58–59, **59**
  manufacturing, 4
Sending overhead
  OCNs *vs.* SANs, **F-27–29**
  time of flight, F-14
Sending overhead, communication latency, I-3
Sense-reversing barrier
  code example, I-15, **I-21**
  large-scale multiprocessor, synchronization, I-14–16
Sequency number, packet header, F-8
Sequential consistency (SC), 417, **421**, 457
  implementation, 418, 423
  programmer's view, 418–419
Sequential interleaving, 100
Sequent Symmetry, M-59–60
Serial Advanced Technology Attachment (SATA) disks
  NetApp FAS6000 filer, D-42
  power consumption, D-5

  RAID 6, D-8–9
  vs. SAS drives, D-5, **D-5**
Serial Attach SCSI (SAS) drive
  historical background, M-88
  power consumption, D-5
  vs. SATA drives, **D-5**
Serialization
  barrier synchronization, I-16
  cache coherence, 378–381
  definition, 380–382, 413
  DSM multiprocessor cache coherence, I-37
Serpentine recording, M-85
Serve-longest-queue (SLQ) scheme arbitration, F-49
Server(s), A-2, 8–9. *See also* Warehouse-scale computer (WSC)
  benchmarks, 43–45
  CPU utilization, **475**
  definition, D-25
  Google WSCs, 512–513, **513**
  single-server model, **D-25**
  system characteristics, **E-4**
Server computer, RISC architectures survey for, K-3–29
Serverless Computing, 496
ServerNet interconnection network, F-70–71
Server utilization, D-25, D-28–29
Service accomplishment, 36
Service interruption, 36
Service level agreements (SLAs), 36–37
Service level objectives (SLOs), 36, 485–486
Session layer, **F-84**
Set associativity, 81
  AMD Opteron data cache, B-13
  cache block, B-8, **B-8**
  cache misses, **B-10**
Set, definition, B-8
Settle time, D-46
SFS benchmark, NFS, D-20–21
Shadow page table, 123
Sharding, 480–482
Shared data, 377
Shared-media networks, F-23–25
Shared memory, 373, 379, 406
  address space, 373
  distributed (*see* Distributed shared memory (DSM))
Shared-memory communication, large-scale multiprocessors, I-4–5
Shared-memory multiprocessors (SMPs), 371, 373
  access time, 371
  definition, M-64
  history background, M-61
  snooping coherence protocols, 380
Shared state
  cache block, 386
  definition, 383–384
Sharing addition, segmented virtual memory, B-52–53
Shear algorithms, disk array, D-51–54
Sheet Generator (SHG), 585
Shell code, 569, **570**
Shifting over zeros, integer multiplication/division, J-45–47
SiFive, **33**
Signals, definition, E-2
Signal-to-noise ratio (SNR), wireless networks, E-21
Signed-digit representation
  example, **J-54**
  integer multiplication, J-53
Signed number arithmetic, J-7–10
Sign-extended offset, RISC, C-5
Significand, J-15
Sign magnitude, J-7–8
Silicon Graphics Altix, M-64
Silicon Graphics Challenge, M-60
Silicon Graphics 4D/240, M-59–60
Silicon Graphics Origin, M-62, M-64
Silicon Graphics systems (SGI), vector processor history, G-27
Simultaneous multithreading (SMT), 424–426, **425**, 435
  definition, 244
  historical background, M-35–36
  implementations, 245
  Java and PARSEC benchmark without, 435–437, **435**, **437**
  multicore processor and, 436–437
  superscalar processors, 245–247
Single chip multicore processor, 382, **391**, 446–451
Single-event upsets (SEUs), 569
Single-extended precision floating-point arithmetic, J-33–34
Single instruction multiple data (SIMD), 11, 170, 282
  historical overview, M-55–57
  instruction
    DSP media extensions, E-10
    IBM Blue Gene/L, I-42
    Sony PlayStation 2, **E-16**
  Intel Core i7 920 multicore computer, **309**
  loop-level parallelism, 170
  multimedia extensions
    256-bit-wide operations, **304**
    data-level parallelism, 304–310
    GPU and MIMD *vs.*, 347–353
    vs. GPUs, 335, **335**
    Intel Core i7 920 multicore computer, **309**
    NEC SX-9 vector processor, **309**
    programming, 307
    RISC-V, 306–307
    roofline visual performance model, 307–310
  NEC SX-9 vector processor, **309**
  processors
    multithreaded, 311, 315, **317**
    Pascal GPU architecture, **330**
    RISC V, 306–307
  supercomputer development, M-45–46
  system area network history, F-104
  thread instructions, 315–317, **316**, **319**
  thread schedule, 315–317
  TI 320C6x DSP, E-9
Single instruction, multiple thread (SIMT), 311
Single instruction stream, single data stream (SISD), 11, M-56
  SIMD, M-46
Single-precision floating point
  arithmetic, J-33–34
  representation, J-15
Single-precision floating-point arithmetic, 329
Single-Streaming Processor (SSP)
  Cray X1, G-21–24
  Cray X1E, G-24
Skippy algorithm, D-49, **D-50**

Small Computer System Interface (SCSI)
  Berkeley's Tertiary Disk project, D-4
  dependability benchmarks, D-21
  disk storage, D-4
  historical background, M-88
  I/O subsystem design, D-59–61
  RAID reconstruction, D-56
  storage area network history, F-106–107
Small form factor (SFF) disk, M-86
Smalltalk, K-21–22
Smart interface cards, *vs.* smart switches, F-90
Smart switches, *vs.* smart interface cards, F-90
SMPs. *See* Shared-memory multiprocessors (SMPs)
SMT. *See* Simultaneous multithreading (SMT)
Snooping bandwidth, 389–390
Snooping cache coherence, 380, **381**
  example protocol, 383–387, **384**
  implementation, 392–393
  invalidate protocol, **381**
  large-scale multiprocessors, I-34–35, M-61
  latencies, 447, **448**
  limitations, 389–392
  maintenance, 380–381
  sample types, **M-60**
SoC. *See* System-on-chip (SoC)
Soft cores, 130
Soft errors, 93
Soft real-time, 7–8
  definition, E-3–4
Software as a service (SaaS)
  growth of, 9
  WSCs, 467
Software guard extensions (SGX), 125
Software pipelining
  example calculations, H-13–14
  loops, execution pattern, **H-15**
  technique, H-12–15, **H-13**
Software prefetching, 148–164
Software speculation
  definition, 176
  hardware-based *vs.*, 240–241
Software technology
  large-scale multiprocessor, I-6
  synchronization, I-17–18
  network interfaces, F-7–8
Solaris, RAID benchmarks, D-21, **D-22**, D-23
Sonic Smart Interconnect, OCNs, F-3
Sony PlayStation 2
  block diagram, **E-16**
  embedded multiprocessors, E-14–15
  Emotion Engine case study, E-15–18
  Emotion Engine organization, **E-18**
Sort procedure, VAX
  code example, K-62–64
  full procedure, K-65
  register allocation, K-62
  register preservation, K-64–65
Source routing, F-49
SPARC "annulling" branch, K-18–19
SPARCLE processor, M-35–36
SPARC v.9
  additional instructions, **K-22**
  fast traps, K-20–21
  integer arithmetic, **K-21**
  LISP, K-21–22
  misaligned trap, **K-21**
  register windows, K-20
  Smalltalk, K-21–22
SPARC VIS, K-25–26, **K-27**
SPARC64 X+, 389, 426, 429
  feature, **427**
  performance, 429–431, **432**
Sparse matrices, vector architecture, G-12–14, 301–302
Spatial locality, B-26
  coining of term, M-11
  computer design principles, 49
  definition, B-2
SPEC benchmark
  active benchmarks, **44**
  correlating predictors, 182
  desktop performance, 41–43, **42**
  early performance measures, M-7
  organization, 433–434
  server performance, 43–45
  static branch prediction, C-22, **C-23**
  storage systems, D-20–21
  vector processor history, G-27–28
SPEC89 benchmark, 41
  branch-prediction buffer, C-24–25, **C-25**
  misprediction rate, **187**
  mispredictions rate, **C-26**
  tournament predictors, 187
SPEC92 benchmarks
  CPI, **C-64**
  stalls, C-61–62
SPEC95 benchmarks
  procedure returns, 232
  return address buffer, 232, **233**
SPEC2000 benchmarks
  compulsory miss rate, B-23
  perl benchmark, **144**
  speculation, 238–239
SPEC2006Cint execution times, **47**
SPECCPU2006 benchmark
  Intel Core i7 920/6700, **192**
  nonblocking caches, 101–102
  virtual machine, 121
SPEC CPU95 benchmark, return address buffer, 232, **233**
SPECCPUint2006 benchmark, clock cycles per instruction, 256, **257**
SPECfp benchmark
  Intel Core i7, 253
  interconnection network, F-91–92
  Itanium 2, **H-43**
  stalls, C-55, **C-56–57**
SPECfpRate benchmark
  cost-performance, 440, **441**
  speedup, 440, **440**
SPEChpc96 benchmark, G-27–28
Special instruction caches, 128
Special-purpose machines
  historical background, M-4–5
  SIMD computer history, M-56–57
Special-purpose register computer, A-3
Special values, floating point, J-14–15, **J-16**
SPECInt2006 benchmark
  ARM Cortex-A53, 132, **132–133**, **250**
  L1 data cache miss rate, **139**
SPECINT92 benchmark, nonblocking caches, 101–102
SPECINT benchmarks
  interconnection network, F-91–92
  Itanium 2, **H-43**
SPECint95 benchmarks, F-92
SPECintRate benchmarks
  cost-performance, 440, **441**

  - performance scaling, 429–431, **431**
  - speedup, 440, **440**

- SPEC Mail benchmark, D-20–21
- SPEC-optimized processors, *vs.* density-optimized, F-89
- SPECpower benchmark, WSCs, 475–476
- SPECRate benchmark, 439
  - for memory-intensive benchmarks, 116, **116**
  - server performance, 43
- SPECRatios, 46–47
- SPEC SFS benchmarks, D-20
- Speculation
  - address aliasing prediction, 239–240
  - advanced techniques, 228–240
  - advantages, 237–238
  - challenge of issues per clock, 236–237
  - concept origins, M-31
  - control dependence, 175–176
  - cross-cutting issues, 127–128
  - disadvantages, 238
  - and energy efficiency, 238–239
  - exception handling, 199
  - execution, 241
  - hardware-based, 208–217
    - data flow execution, 209
    - definition, 208
    - disadvantage, 241
    - instruction execution step, 211–212
    - key ideas, 208
    - reorder buffer, 209–212, 214–215
    - *vs.* software speculation, 240–241
    - write result, 217
  - IA-64, H-38–40
  - ILP studies, M-33–34
  - memory reference, hardware support, H-32
  - microarchitectural techniques case study, 266–273
  - multiple branches, 238
  - register renaming *vs.* ROB, 234–236
  - software, 176, 240–241
- SPEC Web benchmarks, D-20–21
- Speedup
  - Amdahl's Law, 374–375
  - computer design principles, 49–52
  - floating-point addition, J-25–26
  - integer addition
    - carry-lookahead, J-37–41
    - carry-lookahead circuit, **J-38**
    - carry-lookahead tree, **J-40**
    - carry-lookahead tree adder, **J-41**
    - carry-select adder, J-43–44, **J-43–44**
    - carry-skip adder, J-41–43, **J-42**
    - overview, J-37
  - integer division
    - radix-2 division, **J-55**
    - radix-4 division, **J-56**
    - radix-4 SRT division, **J-57**
    - with single adder, J-54–57
    - SRT division, J-45–47, **J-46**, J-55–57
  - integer multiplication
    - array multiplier, **J-50**
    - Booth recoding, **J-49**
    - even/odd array, **J-52**
    - many adders, J-50–54, **J-50**
    - multipass array multiplier, **J-51**
    - signed-digit addition table, **J-54**
    - with single adder, J-47–49, **J-48–49**
    - Wallace tree, **J-53**
  - integer multiplication/division, shifting over zeros, J-45
  - integer SRT division, J-45–47, **J-46**
  - linear, 438–439, **440**
  - from multithreading, **248**
  - pipeline with stalls, C-11–12
  - SPECfpRate benchmarks, 440, **440**
  - SPECintRate benchmarks, 440, **440**
  - switch buffer organizations, F-59–60
  - TPC-C benchmarks, 440, **440**
- Sperry-Rand, M-4–5
- Spin locks, 414–416
  - large-scale multiprocessor synchronization
    - barrier synchronization, I-16
    - exponential back-off, **I-17**
- SPRAM, Sony PlayStation 2 Emotion Engine organization, **E-18**
- Sprowl, Bob, F-103
- Squared coefficient of variance, D-27
- SRAM. *See* Static random-access memory (SRAM)
- SRT division
  - chip comparison, J-61
  - complications, J-45–47
  - early computer arithmetic, J-65
  - example, **J-46**
  - historical background, J-63
  - integers, with adder, J-55–57
  - radix-4, J-56–57, **J-57**
- Stack, A-3, A-28
  - architecture, historical background, M-17–18
- Stacked DRAM, 91
- Stack frame, K-57
- Stack pointer, K-57
- Stale copy, cache coherency, 128
- Stall
  - control dependences, 176
  - cycles
    - average memory access time, B-18
    - branch scheme performance, C-21
    - definition, B-3–4, B-6, **B-22**
    - miss rate calculation, B-6
    - out-of-order execution, B-20
  - data hazards minimization, **C-13**, C-14–15
  - data hazards requiring, C-16–17
  - longer latency pipelines, C-49, **C-50**
  - pipelining performance with, C-11–12
  - RAW, C-49, **C-50**, C-51
  - SPEC92 benchmarks, C-61–62
  - SPECfp benchmarks, C-55, **C-56–57**
- Standardization, commercial interconnection networks, F-67–68
- Standard Performance Evaluation Corporation (SPEC), 41
- Start-up time
  - DAXPY on VMIPS, G-20–21
  - definition, 292
  - page size selection, B-47
  - vector architectures, G-4, **G-4**, **G-8**
  - vector convoys, **G-4**

Start-up time *(Continued)*
  vector performance, G-2–4, G-16
  vector processor, G-7–9, G-25
  VMIPS, **G-5**
Statically based exploitation, ILP, H-2
Static power, 80
Static random-access memory (SRAM)
  arithmetic operations and energy cost, **29**
  memory hierarchy design, 85
  price pressures, 34
  vector memory systems, G-9–10
  vector processor, G-9–10, G-25
Static scheduling
  definition, C-65
  instruction-level parallelism, 218–222
  unoptimized code, C-70
Stencil computation, 550
Sticky bit, J-18
Stochastic gradient descent, 548
Storage area networks, F-77–81, F-106–108
  dependability benchmarks, D-21–23
Storage systems
  Amazon, Dynamo key-value, 485–486
  asynchronous I/O and operating systems, D-35–36
  Berkeley's Tertiary Disk project, D-12–13
  block servers *vs.* filers, D-34–35
  bus replacement, D-34
  component failure, D-43
  computer system availability, D-43
  dependability benchmarks, D-21–23
  dirty bits, D-61–64
  disk array deconstruction, D-51–54
  disk arrays, D-6–10
  disk deconstruction, D-48–50
  disk power, D-5
  disk seeks, D-45
  disk storage, D-2–10
  file system benchmarking, D-20–21
  Internet Archive Cluster (*see* Internet Archive Cluster)
  I/O performance, D-15–23
  I/O system design/evaluation, D-59–61
  mail server benchmarking, D-20–21
  NetApp FAS6000 filer, D-41–43
  operator dependability, D-13–15
  OS-scheduled disk access, D-44
  point-to-point links, D-34
  queuing theory, D-23–34
  RAID performance prediction, D-57–59
  RAID reconstruction case study, D-55–57
  real faults and failures, D-10–15
  reliability, D-15–23
  response time restrictions for benchmarks, **D-18**
  seek distance comparison, **D-47**
  seek time *vs.* distance, **D-46**
  server utilization calculation, D-28–29
  sorting case study, D-64–67
  Tandem Computers, D-13
  throughput *vs.* response time, D-16–18
  TP benchmarks, D-18–20
  transactions components, **D-17**
  web server benchmarking, D-20–21
  WSCs, 478
Store-and-forward packet switching, F-51
Store conditional
  advantage, 414, 416
  definition, 413–414
Store instructions, C-5–6, 199. *See also* Load-store instruction set architecture
Store unit
  bandwidth for, 298–299
  definition, 285
Streaming SIMD Extensions (SSE), 305
Stride, 300
  vector memory systems, G-10–11
Strided accesses, 346
Strided addressing, A-31–32. *See also* Unit stride addressing
Striping, D-51
  disk arrays, D-6
  RAID, D-8
Strip-mined vector loop
  convoys, G-5
  DAXPY on VMIPS, G-20–21
Strip mining, 180, 296, **297**
  DAXPY on VMIPS, G-20–21
Strong scaling, 439
Structural hazards
  check for, C-52
  definition, C-11
Subblocking, cache optimization, 114
Subset property, 423
Sun Microsystems, B-38
Sun Microsystems Enterprise, M-60
Sun Microsystems Niagara (T1/T2), multithreading history, M-35
Sun Microsystems SPARC
  conditional instructions, H-27
  integer arithmetic, J-11–12
  integer overflow, **J-11**
  RISC history, M-21
  synchronization history, M-64–65
Sun Microsystems SPARCCenter, M-60
Sun Microsystems SPARCstation-2, F-92
Sun Microsystems SPARCstation-20, F-92
Sun Microsystems SPARC V8, floating-point precisions, J-33
Sun Microsystems SPARC VIS, multimedia support, **E-11**
Sun Microsystems UltraSPARC, M-63, M-74
Sun Microsystems UltraSPARC T1 processor, **F-78**
Sun Modular Datacenter, M-76
SUN servers, 94
Sun Ultra 5, **47**
Superblock scheduling
  basic process, H-21–23, **H-22**
  compiler history, M-33
Supercomputers, 10
  clusters, **F-80**
  commercial interconnection networks, **F-37**
  direct network topology, **F-37**
  SAN characteristics, F-30–31
  SIMD, development, M-45–46
Superpipelining, C-55
Superscalar processors, 223
  announcement, M-35
  coarse-grained multithreading, 245

  - coining of term, M-31, M-34
  - dynamically scheduled, M-36, 224
  - functional unit execution slots, 244–245, **244**
  - ILP, M-33–34
  - recent advances, M-35
  - simultaneous multithreading, 245–247
- Supervised learning, 547–548
- Supervisor process, virtual memory, 119
- Sussenguth, Ed, M-29–30
- Sutherland, Ivan, M-35
- Swap procedure, VAX, K-57
  - code example, K-59–60
  - full procedure, K-61
  - register allocation for, K-59
  - register preservation, K-60–61
- Switched-media networks, F-2, F-24–25
- Switched networks
  - centralized, F-31–35
  - distributed, F-35–40
- Switches
  - context, B-49
  - early LANs and WANs, F-29
  - interconnecting node calculations, **F-32–33**
  - *vs.* NIC, **F-90**
  - process switch, B-49
  - statements, A-17
  - storage systems, D-34
  - switched-media networks, F-24–25
- Switch fabric, switched-media networks, F-24–25
- Switching, F-21–22, F-44–56
- Switch microarchitecture, basic microarchitecture
- Switch ports, F-30
- Syllable, IA-64, H-35
- Symbolic loop unrolling, software pipelining, H-12–15, **H-13**
- Symmetric multiprocessors (SMP), F-106
  - characteristics, **I-45**
  - first vector computers, M-49
- Symmetric shared-memory multiprocessors, 371
  - limitations, 389–392
  - performance, 393–404
    - commercial workload, 394–399
    - multiprogramming and OS workload, 399–404
  - scientific workloads, I-21–26, **I-23–26**
- Synapse N+1, M-59–60
- Synchronization, 352, 412
  - Cray X1, G-23
  - fetch-and-increment, 413–414
  - hardware primitives, 412–414
  - historical background, M-64–65
  - large-scale multiprocessors
    - barrier synchronization, I-13–16, **I-14**, **I-16**, **I-19**, I-20
    - hardware primitives, I-18–21
    - performance challenges, I-12–16
    - sense-reversing barrier, **I-21**
    - software implementations, I-17–18
    - tree-based barrier, **I-19**
  - locks using coherence, 414–417, **416**
  - message-passing communication, I-5
- Synchronous dynamic random-access memory (SDRAM)
  - capacity and access times, **88**
  - IBM Blue Gene/L, I-42–43
  - memory hierarchy design, 87–90
  - power consumption reduction, 89–90
- Synchronous events, exception, C-39
- Synchronous I/O, definition, D-35
- Synonyms, address translation, B-38
- Synthetic benchmarks, 40
- System area networks, F-76–77, **F-80**, F-104–106
- System call, virtual memory, 119
- System interface controller (SIF), F-74
- System-on-chip (SoC)
  - cell phone, E-24
  - cost trends, 31
  - cross-company interoperability, F-23
  - DSAs, 592–594
  - embedded systems, E-3
  - Sanyo digital cameras, **E-20**
  - Sanyo VPC-SX500 digital camera, E-19
- System response time, D-16
- Systems software, 503
- System/storage area networks (SANs)
  - characteristics, F-3
  - communication protocols, F-8
  - congestion management, F-68–70
  - cross-company interoperability, F-67–68
  - effective bandwidth, F-19
  - fat trees, F-34–35
  - fault tolerance, F-71
  - InfiniBand, F-77–81
  - interconnection network domain relationship, F-4, **F-5**
  - latency and effective bandwidth, F-29–30
  - packet latency, **F-13**, F-14–16
  - time of flight, F-14
- System virtual machines, 120–121
- Systolic array, 560

T

- Tag, 383
  - AMD Opteron data cache, B-13–14
  - memory hierarchy, 81
  - registers, **202**
  - virtual memory fast address translation, B-46
  - write strategy, B-10
- Tag check
  - MIPS R4000 pipeline, C-58–59
  - write strategy, B-10
- Tag field, B-8–9
- Tagged hybrid predictors, 188–190, **188**, **190**
- Tail duplication, superblock scheduling, H-21
- Tailgating, G-20–21
- Tail latency, 473
- Tail tolerant systems, 486
- Tandem Computers, D-13
  - cluster history, M-62, M-74, M-87
- Target address
  - branch hazards, C-18–19
  - branch penalty reduction, C-19–20
  - branch-target buffers, 231
  - pipeline branch issues, C-35–36
  - RISC instruction set, C-5
- Target channel adapters (TCAs), F-90
- Target instructions
  - branch-target buffers, 231
  - GPU conditional branching, 323
- Task-level parallelism (TLP), 10
- TB-80 VME rack, **D-38**, D-41

- Technology trends
  - bandwidth over latency, 20
  - implementation technologies, 19–20
  - scaling of transistor performance and wires, 21–23
- Temporal locality, B-26
  - coining of term, M-11
  - computer design principles, 49
  - definition, B-2
- Tensor processing unit (TPU)
  - architecture, 557–558
  - block diagram, **558**
  - case study, 606–617
  - die, **562**
  - factors limiting, **598**
  - guidelines, 566–567
  - implementation, 560–563
  - improving, 564–566
  - instruction set architecture, 559
  - microarchitecture, 559–560
  - origin, 557
  - printed circuit board, **563**
  - software, 563
  - TensorFlow program, **564**
- TERA processor, M-35
- Terminate event, exception, C-40
- Tertiary Disk project, D-12–13
- Tesla, M-52
- Test-and-set operation, synchronization, 413
- Texas Instruments 8847
  - arithmetic functions, J-57–62
  - chip comparison, **J-58**
  - chip layout, **J-59–60**
- Texas Instruments ASC, first vector computers, M-47
- TFLOPS, parallel processing debates, M-58
- Thacker, Chuck, F-103
- Thermal design power (TDP), 24
- Thin-film transistor (TFT), Sanyo VPC-SX500 digital camera, E-19
- Thinking Machines, M-46, M-56, M-87
- Thinking Multiprocessors CM-5, M-60–61
- Think time, D-16
- Third-level caches, 166, 262
  - interconnection network, F-91–92
- Thrash, B-25–26
- Thread Block, 311, 315
- Thread Block Scheduler, 315, **316**
- Thread-level parallelism (TLP), 5, 10–11, 369
  - centralized shared-memory multiprocessor, 371, 377
    - basic schemes for enforcing coherence, 379–380
    - cache coherence protocol, 377–379, **378**, 383–387, **384**
    - extensions to coherence protocol, 388
    - implementation techniques, 382–383
    - SMP and snooping limitations, 389–392
    - snooping coherence protocols, 380–381, **381**, 392–393
    - structure, **372**
  - definition, 242
  - directory-based cache coherence, 380
    - case study, 451–452
    - protocol example, 408–412
  - distributed shared memory, 371, 373
    - access time, 372–373
    - architecture, **373**
    - directory-based cache coherence, 404–412, **405**
    - disadvantages, 372–373
  - embedded systems, E-15
  - memory consistency, 379, 417–422
    - case study, 456–458
    - compiler optimization, 422
    - programmer's view, 418–419
    - relaxed consistency models, 419–422, **421**
    - speculation to hide latency, 422–423
  - multicore processor, 369, 371–372, 382, 387, 408
    - approaches, 389
    - architecture, **430**
    - coherence, 387
    - development, 404
    - DSM, **373**, **405**, **452**
    - Intel i7 920 performance and energy efficiency, 434–437
    - on multiprogrammed workload, 426–432
    - performance, 426–437, **432**
    - scalability in Xeon E7 with different workloads, 433–434
    - scaling, **432**, 442–444
    - single chip, 382, **391**, 446–451
    - and SMT, 436–437
  - multiprocessor architecture, 370–373
  - *vs.* multithreading, M-36
  - parallel processing challenges, 373–377
  - single-chip multicore processor, 446–451
  - synchronization, 412
    - hardware primitives, 412–414
    - locks using coherence, 414–417, **416**
- Thread of SIMD instructions
  - GPU programming, 315–317, **316**
  - scheduling, **319**
- Three-dimensional space, direct networks, F-39
- Throttling packets, F-10
- Throughput, 20, 39. *See also* Bandwidth
  - computing kernel, 350, **351**
  - definition, C-3, F-13
  - disk storage, **D-4**
  - DNN applications, 596–600
  - producer-server model, D-15–16, **D-16**
  - *vs.* response time, D-16–18
  - routing comparison, **F-54**
  - uniprocessor, 242–247
- Thumb-2, K-3
  - 16-bit instructions, **K-7**, **K-10**
  - register encodings, **K-7**
- Tilera TILE-Gx processors, OCNs, F-3
- Time, and cost, 30–31
- Time-constrained scaling, I-33–34
- Time division multiple access (TDMA), cell phones, E-25
- Time of flight
  - communication latency, I-3
  - interconnection networks, F-14
- Time-sharing, B-49–50
- Timing independent, M-18

TI TMS320C55 DSP
  architecture, **E-7**
  characteristics, E-7–8
  data operands, **E-6**
TI TMS320C6x DSP
  architecture, **E-9**
  characteristics, E-8–10
  instruction packet, **E-10**
TLB. *See* Translation lookaside buffer (TLB)
TLP. *See* Thread-level parallelism (TLP)
Tomasulo's algorithm
  advantages, 201
  definition, 194–195
  dynamic scheduling, 195–201
  RAW, 217
  RISC-V floating-point unit, **198**
  steps in, **216**
TOP500, M-59
Top of Rack (ToR) switch, 477–478
Topology, F-21–22, F-30–44
Torus networks, **F-53–56**, F-76–77
Total cost of ownership (TCO), 577
  case study, 519–521
  DSAs, 600–601, **601**
  resource allocation, 521–522
Total store ordering (TSO), 420, **421**
Tournament predictors, 184–188, **186**
  advantage, 185–187
  branch address, **186**
  early schemes, M-29
  local/global predictors, 184–188, **186**
Toy programs, 40
TPC-C benchmarks
  definition, 44, 439
  speedup, 440, **440**
TPC-C, file system benchmarking, D-18–20
TPU. *See* Tensor processing unit (TPU)
Trace compaction, H-19
Trace scheduling, H-19–21, **H-20**
Trace selection, definition, H-19
Traffic intensity, queuing theory, D-26
Trailer
  messages, F-6
  packet format, **F-7**
Transaction components, D-16, I-38–39
Transaction-processing (TP)
  benchmarks, server performance, 43–44
  storage system benchmarks, D-18–20
Transaction Processing Council (TPC), 43–45
  benchmarks overview, D-18–20
Transfers, A-16. *See also* Data transfers
Transforms, DSP, E-5
Transient failure, F-70
Transient faults, D-11, 93
Transistor performance, scaling, 21–23
Translation lookaside buffer (TLB)
  address translation, B-37, B-46, **B-47**
  ARM Cortex-A53, 251–252
  coining of term, M-9
  interconnection network protection, F-91
  misses, 346
  Opteron, **B-47**, B-56–57
  speculation, 237–238
Transmission Control Protocol (TCP), congestion management, F-69
Transmission Control Protocol/Internet Protocol (TCP/IP), F-86
  ATM, F-102–103
  headers, **F-88**
  internetworking, F-85–89
  reliance on, F-99
  WAN, F-102
Transmission speed, interconnection network performance, F-13
Transmission time, F-14
  communication latency, I-3
Transport latency
  time of flight, F-14
  topology, F-25–26
Transport layer, **F-84**
Transputer, F-105
Trap-handling routines, C-54
Tree-based barrier, large-scale multiprocessor synchronization, **I-19**
Tree height reduction, H-11
Trellis codes, definition, E-6–7
TRIPS Edge processor, F-67
Trojan horses, B-51–53
True dependence, finding, H-7–8
True sharing misses, 393–394, 397, **398**
TSMC, Stratton, F-3
TSO. *See* Total store ordering (TSO)
TSS operating system, M-9
Turbo mode in 2008, 28
Turing, Alan, M-4, M-20
Turn Model routing algorithm, F-48
Two-dimensional line buffer, 589–590
Two-level predictors, 183, 191
Two's complement, J-7–8
Two-way set associativity, B-8
  average memory access time, B-19
  conflict misses, B-23
  Opteron data cache, B-13–14, **B-13**
  2:1 cache rule of thumb, B-29
TX-2, M-35, M-50

U

Ultrix, DECstation 5000 reboots, **F-73**
UMA. *See* Uniform memory access (UMA)
Unbiased exponent, J-15
Uncached state, 406
Underflow
  floating-point arithmetic, J-36–37, J-62
  gradual, J-15, J-36
Unicasting, shared-media networks, F-24
Unified buffer, 558
Unified cache
  AMD Opteron example, B-15, **B-15**
  miss rate, B-16
Unified virtual memory, 330
Uniform memory access (UMA), 371
Uninterruptible power supply (UPS), 504
Uniprocessor, 377–378
  cache coherence mechanism, **384**, 386
  throughput, 242–247
Unit stride addressing, A-31–32
UNIVAC I, M-5, M-17
UNIX systems, B-38
  block servers *vs.* filers, D-34
  floating point remainder, J-32
  miss statistics, **B-59**
  seek distance comparison, **D-47**

UNIX systems *(Continued)*
  vector processor history, G-26
  workload, 399
Unoptimized code, C-70
Unpacked decimal, A-14, J-16
Unshielded twisted pair (UTP), F-104
Up*/down* routing, F-49
USB, Sony PlayStation 2 Emotion Engine case study, E-15
Use bit, B-45–46, B-52
User-level communication, F-8
User maskable events, exception, C-39
User nonmaskable events, exception, C-39
User requested events, exception, C-39
User Space Driver, 563
Utility computing, M-75–76
Utilization
  I/O system calculations, D-26
  queuing theory, D-25

V

Valid bit, B-8, 383, 393–394
  address translation, B-46
  AMD Opteron data cache, B-14
  page table entry, B-52
Value prediction, 228, 234
VAPI, InfiniBand, F-81
Variable length, 14
Variables, random, distribution, D-26–34
VAX architecture
  fallacies and pitfalls, K-65–67
  instructions encoding, K-54–55
  operands and addressing modes, K-51–54
  operations, K-56–57
  sort, K-62–65
  swap, K-59–61
Vector architecture, 10, A-31, 282
  computer development, M-45–46
  execution time, 290–293
  fallacy, 356
  vs. graphics processing units, 331–334
  memory banks, 298–299
  memory systems, G-9–11
  multidimensional arrays, 299–301
  multiple lanes, 293–294
  pitfall, 355–356
  predicate registers, 296–298
  processor example, 288–290
  programming, 302–304
  RV64V extension, 283–287, **284**
  sparse matrices, 301–302
  start-up latency and dead time, **G-8**
  vector-length registers, 294–296
  vector-register characteristics, **G-3**
Vector array, 585
Vector element, 289
Vector functional units, 285
Vector instruction
  definition, 289
  instruction-level parallelism, 170
Vectorized code, 289
Vectorizing compilers
  effectiveness, G-14
  FORTRAN test kernels, **G-15**
  sparse matrices, G-12–13
Vector kernel implementation, case study, 357–359
Vector-length register (VLR), 294–296
  performance, G-4–5
Vector load
  bandwidth for, 298–299
  definition, 285
Vector-mask control, 297
Vector-mask registers
  Cray X1, G-21–22
Vector processor
  Cray X1, G-21–24, **G-22–23**
  Cray X1E, G-24
  DAXPY on VMIPS, G-17, G-19–21
  definition, 370
  DSP media extensions, E-10
  execution time, **G-7**
  historical background, G-26–28
  measures, G-15–16
  NEC SX-9 vector processor, **309**
  overview, G-25–26
  performance, G-2–9
    chaining, G-11–12, **G-12**
    DAXPY on VMIPS, G-17
    sparse matrices, G-12–14
    start-up and multiple lanes, G-7–9
    unchaining, **G-12**
  vs. scalar processor, G-19
  Sony PlayStation 2 Emotion Engine, E-17–18
  start-up overhead, **G-4**
  vector kernel implementation, 357–359
  VMIPS on DAXPY, G-17, G-19–21
  VMIPS on Linpack, G-17–19
Vector registers, 284
Very-large-scale integration (VLSI)
  early computer arithmetic, J-63
  interconnection network topology, F-30
  RISC history, M-21
  Wallace tree, J-52–53
Very long instruction word (VLIW)
  compiler history, M-32
  EPIC approach, M-33
  IA-64, H-33
  instruction set, 587, **587**
  multiple issue processors, 218–222, **220**, **271**
  multiple-issue processors, M-30
  multithreading history, M-36
  TI 320C6x DSP, E-8–10
VGA controller, M-51
VI interface, M-63, M-74
Virtual address
  AMD Opteron data cache, B-12–13
  memory hierarchy, **B-39**
  miss rate *vs.* cache size, **B-37**
  Opteron mapping, **B-55**
  Opteron memory management, B-54–57, **B-55**
  and page size, B-58
  physical address, **B-45**
  translation, B-36–40
  virtual memory, **B-41**, B-42, B-44
Virtual cache, B-36–38
Virtual channels (VCs), F-47–48
  HOL blocking, **F-60**
  switching, F-52
  switch microarchitecture pipelining, F-66
  system area network, F-105–106
  and throughput, F-97
Virtual cut-through switching, F-52
Virtual functions, A-17
Virtual instruction set architecture (VISA), 587–588
Virtualization
  Intel 80x86 instruction, **145**
  memory hierarchy design, 126–127

Virtual Machine Control State (VMCS), 146
Virtual machine monitor (VMM), 121
  instruction set extension, 124–125
  laissez faire attitude, 145
  pitfall, 145
  requirements, 122
  Xen virtual machine, 126
Virtual machines (VMs), 491
  early IBM work, M-10
  hardware management, 121
  impact on virtual memory, 123–124
  instruction set architecture for, 122–123
  protection via, 120–122
  software management, 121
Virtual memory, B-2–3, B-40–49
  address space, B-12, **B-41**, B-44, B-55
  address translation, B-46, **B-47**
  caches and, B-42–43, **B-42**, B-48–49, **B-48**
  classes, B-43
  instruction set extension, 124–125
  Intel Pentium *vs.* AMD Opteron, B-57
  paged example, B-54–57
  page size selection, B-46–47
  parameter ranges, **B-42**
  Pentium *vs.* Opteron protection, B-57
  protection, B-49–50, 119–120
  questions, B-44–46
  segmented example, B-51–54
  virtual machine, 123–124
Virtual output queues (VOQs), F-60–61
VLIW. *See* Very Long Instruction Word (VLIW)
VLR. *See* Vector-length register (VLR)
VLSI. *See* Very-large-scale integration (VLSI)
VME rack, D-37–38, **D-38**
VMIPS
  DAXPY, G-18–21
  enhanced, DAXPY performance, G-19–21
  peak performance on DAXPY, G-17
  performance, G-4
    on Linpack, G-17–19
  sparse matrices, G-13
  start-up penalties, **G-5**
  vector execution time, G-6–7
  vector performance measures, G-16
Voltage regulator controller (VRC), F-74
Volume, and cost, 30–31
Von Neumann computer, M-2–3
Von Neumann, John, M-2–5
Voodoo2, M-51

W

Wafer
  definition, 31
  RISC-V dies, **33**
  yield, 34
Waiting line, D-25
Wallace tree
  example, J-52–53, **J-53**
  historical background, J-63
Wall-clock time, 39
  scientific applications on parallel processors, I-33
WAR. *See* Write after read (WAR)
Warehouse-scale computer (WSC), 4–5, 9–10, 369–370, 466
  active *vs.* inactive low power modes, 516
  average memory latency, 480
  capital costs, 516
  case study, **487**
  cloud computing
    advantages, 490
    AWS (*see* Amazon Web Services (AWS))
    economies of scale, 491
    fallacy, 514
  cluster history, M-74–75
  computer architecture of, 477–482
  cost, 486–490
  cost-performance, 515, 517
  cost trends, 36
  efficiency
    and cost, 482–490
    energy, 503
    measuring, 483–486
  fault tolerance, 516
  Google
    cooling, 506–508
    networking, 510–511
    power distribution, 504–506
    racks, 509–510
    servers, 512–513
  hierarchy of switches, **477**
  Layer 3 network, **481**
  low-power servers, 519–521
  memory hierarchy, 479–482
  microsecond delays, 517
  opportunities/problems, 468
  performance, 514
  power utilization effectiveness, 483, **484**
  preventing, 501–503
  programming models and workloads, 471–476
  resource allocation, 521–522
  server cost and power, 519–521
  storage, 478
  total cost of ownership, 519–521
Warp, M-32
Water-side economization, 508
Wavelength division multiplexing (WDM), F-103
WAW. *See* Write after write (WAW)
Way prediction, hit time, 98–99
WB cycle. *See* Write-back (WB) cycle
WCET. *See* Worst-case execution time (WCET)
Weak ordering, 420, **421**
Weak scaling, 439
Web servers
  benchmarking, D-21
  WAN, F-102
Weighted arithmetic mean time, D-27
Weight FIFO, 558
Weight memory, 558
Weitek 3364
  arithmetic functions, J-57–61
  chip comparison, **J-58**
  chip layout, **J-59–60**
West-first routing, F-48
Wet-bulb temperature, 508
Whirlwind project, M-4
Wide area networks (WANs)
  ATM, F-4
  characteristics, F-4
  cross-company interoperability, F-68
  effective bandwidth, F-19
  fault tolerance, F-71–73
  historical overview, F-102–103
  InfiniBand, F-77–78

Wide area networks (WANs) *(Continued)*
  interconnection network domain relationship, F-4, **F-5**
  latency and effective bandwidth, **F-27–29**
  offload engines, F-8
  packet latency, **F-13**, F-14–16
  switching, F-51
  time of flight, F-14
Wilkes, Maurice, M-3
Winchester disk design, M-86
Window, F-69
  TCP/IP headers, **F-88**
Wireless networks
  basic challenges, **E-21**
  and cell phones, E-21–22
Wires, scaling of, 21–23
Within instructions exception, C-39
  instruction set complications, C-45
  stopping/restarting exception, C-41
Word(s)
  AMD Opteron data cache, B-14–15
  double, A-7, **A-8**, **A-14**, **A-44**, 300
  DSP, E-6
  half, A-8, **A-8**, **A-14**, **A-44**
Word count, **B-53**
Word displacement addressing, K-52
Word offset, C-28
Working set effect, I-24
Workload, 39–40
  commercial, 394–399
  measurements, 400
  multiprogramming and OS, 399–404
  phases, 399–400
  RAID performance prediction, D-57–59
  scalability in Xeon E7 with, 433–434
  symmetric shared-memory multiprocessors, I-21–26, **I-23–26**
  warehouse-scale computers, 471–476
Wormhole switching, F-52, F-97, F-105
Worst-case execution time (WCET), E-4
Write after read (WAR)
  dynamic scheduling, 193
  hazard, C-12, C-69
  multiple-issue processors, M-30
  program order, 174
  register renaming, 196
  TI TMS320C55 DSP, E-8
  Tomasulo's algorithm, 195, 207
Write after write (WAW), C-12
  check for, C-52
  dynamic scheduling, 193
  longer latency pipelines, C-49, C-51
  multiple-issue processors, M-30
  program order, 173
  register renaming, 196
  Tomasulo's algorithm, 195
Write allocate, B-11–12
Write-back cache, B-11–12
  cache coherence protocol, 385, **385**
  directory-based cache coherence protocol, 411
  memory hierarchy, 81
  snooping coherence, 380–384, **381**
  uniprocessor, 386
Write-back (WB) cycle
  data hazards stall minimization, C-13–14
  MIPS R4000 pipeline, C-58–59
  multicycle FP operations, C-52
  RISC classic pipeline, C-8
  RISC exception, C-43
  RISC instruction set, C-5, **C-6**
  RISC pipeline, C-33
  RISC pipeline control, C-35, **C-36**
  simple RISC implementation, C-29
Write broadcast protocol, 381
Write buffer, B-11, B-14, 382
  memory hierarchy, 81
  merging, 105–106, **106**
Write hit
  cache coherence, **384–385**, 386, **387**
  definition, B-11
Write invalidate protocol, 380
  example, 385, **385**
  implementation, 382–383
  snooping coherence, **381**
Write miss, 411, 418
  AMD Opteron data cache, B-12
  cache coherence, **384–385**, 385, **387**
  directory-based cache coherence protocol, 411
  memory stall clock cycles, B-4
  miss penalty reduction, B-35–36
  operation, 408
  Opteron data cache, B-12, B-14
  options, B-11
Write result, 199
  dynamic scheduling with scoreboard, C-69
  hardware-based speculation, 217
  instruction step, 211
Write serialization
  cache coherence, 378–379
  definition, 380–382, 413
Write stall, B-11
Write strategy
  memory hierarchy, B-45–46
  virtual memory, B-45–46
Write-through cache, B-11–12
  average memory access time, B-16–17
  coherence protocol, **378**, 382–384
  memory hierarchy, 81
Write update protocol, 381

X

XALANCBMK benchmarks, 138
XBox, M-51–52
Xen virtual machine, 126
Xeon E7, 389, 426–429
  feature, **427**
  on-chip organizations, **428**
  performance, 431, **432**
  QuickPath Interconnect, 429
  scalability, 433–434, **434**
Xerox Palo Alto Research Center, F-103
XIMD architecture, M-36
Xon/Xoff, interconnection networks, F-10–11, F-18

Z

Zero-copy protocols, F-8
Zero-load latency, F-75
Z-80 microcontroller, cell phones, E-24
Zuse, Konrad, M-4–5

Translation between GPU terms in book and official NVIDIA and OpenCL terms.

Type	More Descriptive Name used in this Book	Official CUDA/ NVIDIA Term	Book Definition and OpenCL Terms	Official CUDA/NVIDIA Definition
Program Abstractions	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of 1 or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range."	A Grid is an array of Thread Blocks that can execute concurrently, sequentially, or a mixture.
	Body of Vectorized Loop	Thread Block	A vectorized loop executed on a "Streaming Multiprocessor" (multithreaded SIMD processor), made up of 1 or more "Warps" (or threads of SIMD instructions). These "Warps" (SIMD Threads) can communicate via "Shared Memory" (Local Memory). OpenCL calls a thread block a "work group."	A Thread Block is an array of CUDA threads that execute concurrently together and can cooperate and communicate via Shared Memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid.
	Sequence of SIMD Lane Operations	CUDA Thread	A vertical cut of a "Warp" (or thread of SIMD instructions) corresponding to one element executed by one "Thread Processor" (or SIMD lane). Result is stored depending on mask. OpenCL calls a CUDA thread a "work item."	A CUDA Thread is a lightweight thread that executes a sequential program and can cooperate with other CUDA threads executing in the same Thread Block. A CUDA thread has a thread ID within its Thread Block.
Machine Object	A Thread of SIMD Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a "Streaming Multiprocessor" (multithreaded SIMD processor). Results stored depending on a per element mask.	A Warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD processor.
	SIMD Instruction	PTX Instruction	A single SIMD instruction executed across the "Thread Processors" (SIMD lanes).	A PTX instruction specifies an instruction executed by a CUDA Thread.
Processing Hardware	Multithreaded SIMD Processor	Streaming Multiprocessor	Multithreaded SIMD processor that executes "Warps" (thread of SIMD instructions), independent of other SIMD processors. OpenCL calls it a "Compute Unit." However, CUDA programmer writes program for one lane rather than for a "vector" of multiple SIMD lanes.	A Streaming Multiprocessor (SM) is a multithreaded SIMT/SIMD processor that executes Warps of CUDA Threads. A SIMT program specifies the execution of one CUDA thread, rather than a vector of multiple SIMD lanes.
	Thread Block Scheduler	Giga Thread Engine	Assigns multiple "Thread Blocks" (or body of vectorized loop) to "Streaming Multiprocessors" (multithreaded SIMD processors).	Distributes and schedules Thread Blocks of a Grid to Streaming Multiprocessors as resources become available.
	SIMD Thread Scheduler	Warp Scheduler	Hardware unit that schedules and issues "Warps" (threads of SIMD instructions) when they are ready to execute; includes a scoreboard to track "Warp" (SIMD thread) execution.	A Warp Scheduler in a Streaming Multiprocessor schedules Warps for execution when their next instruction is ready to execute.
	SIMD Lane	Thread Processor	Hardware SIMD Lane that executes the operations in a "Warp" (thread of SIMD instructions) on a single element. Results stored depending on mask. OpenCL calls it a "Processing Element."	A Thread Processor is a datapath and register file portion of a Streaming Multiprocessor that executes operations for one or more lanes of a Warp.
Memory Hardware	GPU Memory	Global Memory	DRAM memory accessible by all "Streaming Multiprocessors" (or multithreaded SIMD processors) in a GPU. OpenCL calls it "Global Memory."	Global Memory is accessible by all CUDA Threads in any Thread Block in any Grid. Implemented as a region of DRAM, and may be cached.
	Private Memory	Local Memory	Portion of DRAM memory private to each "Thread Processor" (SIMD lane). OpenCL calls it "Private Memory."	Private "thread-local" memory for a CUDA Thread. Implemented as a cached region of DRAM.
	Local Memory	Shared Memory	Fast local SRAM for one "Streaming Multiprocessor" (multithreaded SIMD processor), unavailable to other Streaming Multiprocessors. OpenCL calls it "Local Memory."	Fast SRAM memory shared by the CUDA Threads composing a Thread Block, and private to that Thread Block. Used for communication among CUDA Threads in a Thread Block at barrier synchronization points.
	SIMD Lane Registers	Registers	Registers in a single "Thread Processor" (SIMD lane) allocated across full "Thread Block" (or body of vectorized loop).	Private registers for a CUDA Thread. Implemented as multithreaded register file for certain lanes of several warps for each thread processor.

RV64G Instruction Subset

Mnemonic	**Function**
Data transfer	*Move data to/from GPRs and FPRs*
`lb,lbu,lh,lhu,lw,lwu`	Load byte, half word, or word to lower portion of GPR with/without sign extension
`ld,sd`	Load or store a double word to GPR
`sb,sh,sw`	Store a byte, half word, or word from lowest portion of GPR to memory
`fld,flw,fsd,fsw`	Load or store a double word or word to/from the FPRs
ALU Operations	*Register-register and register immediate ALU operations*
`add, addi,addw,addiw`	Add, add immediate, add word, or add word immediate. Word version affects lower 32 bits.
`and,andi,or,ori,xor,xori`	AND, AND immediate, or OR immediate, exclusive OR, exclusive OR immediate
`auipc`	Add upper immediate to PC; puts sum of a shifted immediate and PC in a register
`lui`	Loads an immediate value into the upper portion of a word.
`mul,mulw,mulh,mulhsu, mulhu`	Multiply, multiply word, multiply halfword, multiply upper half, signed and unsigned. Word affects lower 32 bits.
`div,diw,divu`	Divide, divide word, divide unsigned.
`rem,remw,remu,remuw`	Remainder, remainder word, remainder unsigned.
`sll,slli,srl,srli,sra,srai`	Shift left /right logical, right arithmetic, immediate and with shift amount in a GPR.
`sllw,sllwi,srlw,srlwi, sraw, sraiw`	Word shifts: affect only the lower 32-bits or a GPR.
`slt,slti,sltiu,sltu`	Set Less then: if first operand less than the second, set destination to 1 else 0; immediate form and signed/unsigned.
`sub,subi,subw,subwi`	Subtract, subtract immediate. Word version affects lower 32 bits.
Control Transfer	*Branches, jumps, procedure calls*
`beq,bge,bgeu,blt,bltu,bne`	Compare two registers if condition is true branch to PC + offset
`jal,jalr`	Jump, Jump to register contents. The address of the next instruction is saved in designated register. Unconditional jump without link by setting destination register to x0.
Floating Point Operations	*Floating point instructions operating of FPRs.*
`fadd.*, fsub.*, fmul.*, fdiv.* ,fsrt.*`	FP add, subtract, multiply, divide, and square root; single (.s) and double (.d) precision versions.
`fmadd.*, fmsub.*, fmnadd.*, fmnsub.*`	Multiply-add, multiply-subtract, negte multiply-add, negate multiply-subtract; single (.s) and double (.d) precision versions.
`fsgnj.*, sgnjn.*, fsgnjx.*`	Copy sign, inverse sign, or XOR of sign to first operand; single (.s) and double (.d) precision versions.
`fmin.*, fmax.*`	Minimum and maximum of two values; single (.s) and double (.d) precision versions.
`feq.*, flt.*, fle.*`	Floating point compares; single (.s) and double (.d) precision versions.
`fclass.*`	Classify type of FP value; single (.s) and double (.d) precision versions.
`fmv.*.x, fmv.x.*`	Move from/to GPRs; single (.s) and double (.d) precision versions.
`fcvt.d.s, fcvt.s.d`	Convert SP to DP or DP to SP
`fcvt.*.w, fcvt.*.wu, fcvt.*.i, fct.*.lu`	Convert from word or double word, signed or unsigned to DP or DP.
`fcvt.w.*, fcvt.wu.*, fcvt.i.*, fcvt.lu.*`	Convert to word or double word, signed or unsigned.

COMPUTER ARCHITECTURE

Sixth Edition

A Quantitative Approach

John L. Hennessy | David A. Patterson

Foreword by Norman P. Jouppi

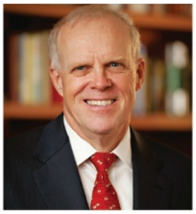
"This sixth edition comes at a critical time: Moore's Law is fading just as deep learning demands unprecedented compute cycles. The new chapter on domain-specific architectures documents a number of promising approaches and prophesies a rebirth in computer architecture. Like the scholars of the European Renaissance, computer architects must understand our own history, and then combine the lessons of that history with new techniques to remake the world."
—Cliff Young, Google

Computer Architecture: A Quantitative Approach, Sixth Edition has been considered essential reading by instructors, students, and practitioners of computer design for nearly 30 years. The sixth edition of this classic textbook is fully revised with the latest developments in processor and system architecture. It now features examples from the RISC-V ("RISC Five") instruction set architecture, a modern RISC instruction set developed and designed to be a free and openly adoptable standard. It also includes a new chapter on domain-specific architectures and an updated chapter on warehouse-scale computing that features the first public information on Google's newest WSC. True to its original mission of demystifying computer architecture, this edition continues the longstanding tradition of focusing on areas where the most exciting computing innovation is happening, while always keeping an emphasis on good engineering design.

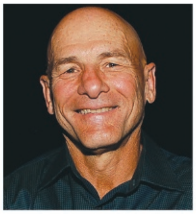
Features

- Includes a new chapter on domain-specific architectures, explaining how they are the only path forward for improved performance and energy efficiency given the end of Moore's Law and Dennard scaling
- Features the introduction of four DSAs from industry: Google Tensor Processing Unit, Google Pixel Visual Core, Intel Nervana Neural Network Processor, and Microsoft Catapult
- Features extensive updates to the chapter on warehouse-scale computing, with the first public information on the newest Google WSC
- Offers updates to other chapters including new material dealing with the use of stacked DRAM; data on the performance of new NVIDIA Pascal GPU vs. new AVX-512 Intel Skylake CPU; and extensive additions to content covering multicore architecture and organization

About the Authors



John L. Hennessy
Professor of Electrical Engineering
and Computer Science
President Emeritus
Stanford University



David A. Patterson
Distinguished Engineer, Google
Pardee Chair of Computer Science,
Emeritus
University of California at Berkeley

Computer Systems and Design
Computer Engineering

ISBN 978-0-12-811905-1

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
elsevier.com/books-and-journals